MASTER EN SCIENCES INFORMATIQUES

Génération dynamique de XML avec un langage de template extensible

De Buyser, Bastien

Award date:
2007

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique.
Année académique 2006 - 2007

Génération dynamique de XML avec un langage de template extensible

Bastien DE BUYSER

Mémoire présenté en vue de l'obtention du grade de maître en informatique

Résumé

L'hétérogénéité des systèmes informatiques présents aux quatre coins de la planète est un fait : ils tournent sur des plateformes différentes, interviennent dans des contextes différents et recourent à des technologies différentes. Avec l'apparition d'Internet, de nombreux systèmes informatiques (des réseaux jusqu'alors indépendants) ont été connectés entre eux. Ce regroupement à l'échelle mondiale ne s'est pas réalisé sans soucis étant donné l'incompatiblité des différentes parties intervenantes. Pour lutter contre ce problème, des organismes publient des standards qui servent de référence commune (élaboration de protocoles de communication, etc.). C'est dans ce contexte qu'intervient XML, un langage de balisage indépendant de toute technologie, réglementé par le W3C.

Sa neutralité favorise l'interopérabilité entre les systèmes informatiques mais il représente aussi un moyen clair et bien défini de structurer de l'information quel que soit le contexte. C'est ainsi qu'au-delà des frontières du Web, les documents XML se retrouvent utilisés dans de nombreuses applications. Le langage est purement statique et totalement inactif, pourtant il est parfois utilisé dans un environnement changeant. Dans certaines circonstances le dynamisme doit être pris en considération, c'est pourquoi des technologies permettant de générer dynamiquement du XML on fait leur apparition. Parmi elles, citons JSP, ASP, CGI ou encore XSLT.

MSP (Mercury Server Page) rentre également dans cette catégorie de technologies. Il est constitué d'un langage de template générique (accompagné de son compilateur) qui permet de générer du XML avec dynamisme. Le projet MSP, qui est l'objet cible de ce mémoire, a été développé chez *Mission Critical*. Cette entreprise belge souhaitait créer un langage de template personnalisé et extensible qui puisse s'intégrer parfaitement à son activité, à savoir le développment logiciel sur base de programmation déclarative.

Abstract

Heterogeneity of IT systems all around the world is a fact : they are run on different platforms, technologies, work in different contexts and address different requirements. With the rise of the Internet, numerous IT systems (previously functioning on independent networks) have been interconnected. This world-wide regrouping did not take place without problems given the different parties involved. In order to reduce this problem, standardisation was required to act as a common reference point for interoperability (creation of communication protocols, etc...). In this context of standardisation and simplification of communications protocols XML arrived. This simple markup language is regulated by W3C and is technologically neutral.

The advantage of XML as a technology neutral communications format makes it ideal as a clear and well defined means of representing and exchanging information whatever the technological context. This neutrality has also meant that XML is not just used within web-based applications, but has also found application outside of web applications. The language is static and fully inactive however it is sometimes used in a changing environnement. In some circumstances the dynamism must be taken into account, it's the reason why technologies enabling to generate XML dynamically have been created. Among them stand JSP, ASP, CGI and XSLT.

MSP (Mercury Server Page) belongs to this category of application level technologies. It is a generic template language (and associated compiler) that enables the dynamic generation of XML. The MSP project, which is the subject of this report, has been developped by *Mission Critical*. This Belgian enterprise wanted to create an extensible template language that could perfectly integrate with its declaravite software development methodology.

Remerciements

La première personne à remercier dans le cadre de ce mémoire est je crois Wim Vanhoof, mon promoteur, à qui je dois l'obtention de mon stage et la supervision de la rédaction qui a suivi.

Le développement de MSP (auquel j'ai participé durant quatre mois) s'est déroulé dans les murs de l'entreprise *Mission Critical*. L'accueil qui m'y a été réservé peut sans nul doute être qualifié de chaleureux, tout en restant professionnel. Je souhaiterais remercier chaque personne que j'y ai rencontrée (même si je ne les citerai pas toutes) pour la sympathie qui a été tenue à mon égard et pour les conseils que chacun a pu m'adresser. Je retiendrai plus particulièrement Michel Vanden Bossche (dirigeant de l'entreprise) qui m'a accepté comme stagiaire, Maxime Van Assche et Ludovic Langevine qui m'ont épaulé durant tout le stage et encore lors de la rédaction du mémoire, Paul Massey et Eric Paesmans pour leurs réponses à mes nombreuses questions, Thomas Fazekas qui a résolu à maintes reprises les problèmes réseaux que j'ai pu rencontrer, Bert Van Nuffelen pour son aide lorsque j'en avais besoin, Rudradeb Mitra avec qui j'ai mené des conversation intéressantes et agréables (notamment lors de nos trajets en bus), et enfin Vinciane Capelle, dont la gentillesse et l'accueil me laisseront des souvenirs bien plaisants de cette expérience à Erps-Kwerps.

La rédaction du mémoire a également nécessité des relectures. A ce sujet, je remercie mes parents (Jacques De Buyser et Patricia Gruselle) ainsi que Myriam Duckers.

Enfin, j'adresse mes remerciements à toutes les personnes qui de près ou de loin m'ont soutenu ou conseillé dans l'épreuve que représente ce travail.

Table des matières

Table des figures

Chapitre 1

Introduction

Tout qui possède un certain niveau de connaissance en informatique (et plus spécialement dans le domaine de l'Internet) a forcément été confronté au géant du W3C qu'est XML. Ce langage apparaît dans une multitude de circonstances mais l'usage le plus courant qui en est fait réside dans le contexte du Web.

Les pages disponibles sur la toile de l'Internet peuvent être séparées en deux catégories : les pages statiques et les pages dynamiques. Le contenu d'une page statique est entièrement prédéfini par son auteur, tous les utilisateurs (peu importe leur localisation, âge, environnement...) en retireront exactement le même résultat. Les pages dynamiques, quant à elles, sont constituées (au moins partiellement) d'éléments qui dépendent d'un contexte changeant. Selon les évènements et les circonstances, les résultats obtenus des requêtes de pages dynamiques diffèreront les uns des autres. Un exemple de page statique est une page écrite entièrement en HTML (qui présente par exemple un article avec quelques références, une recette de cuisine, ou...), la page renvoyée sera toujours la même. Une page dynamique pourrait par exemple, avant d'envoyer le résultat, s'informer sur la langue de l'utilisateur pour lui transmettre la page dans sa langue maternelle. Le dynamisme des pages permet ainsi en ce sens de les personnaliser. Si le contexte du Web est un endroit où on parle beaucoup de XML dynamique, ce n'est pas pour autant le seul cas où on l'utilise. Le chapitre "*1. XML en bref*" ouvrira les portes sur le monde du XML et "*3. Quelques solutions existantes*" traitera certains langages utilisés dans le domaine de la production dynamique de documents XML.

Grâce à sa grande neutralité technologique, ce langage s'ouvre à des horizons très larges. Il existe ainsi des langages de normes XML (tel que le XHTML) qui permettent de créer des interfaces graphiques (aussi appelées UI pour "User Interface"). Si des langages comme le XHTML sont à la base conçus pour intervenir dans le contexte du Web, rien n'empêche de les intégrer dans des applications détachées de leur environnement natal. Ainsi, le XML peut aussi apparaître dans le cadre du développement logiciel où les professionnels de l'IT sont amenés

à concevoir des interfaces graphiques pour interagir avec l'utilisateur. Lors de cette conception, on tente de rendre l'interface "user friendly", fonctionnelle et ergonomique [25] ; ce qui constitue le travail du designer d'interface.

Cependant, si l'interface n'est pas purement statique cela risque de perturber la manière dont elle est générée : l'aspect dynamique obligera souvent à placer l'information sur la génération de l'interface au sein d'un agrégat de lignes de code (rédigées dans un langage de programmation quelconque). Le langage d'interface confondu dans une mêlée de code étranger rend la compréhension et l'exploitation de cette source difficile (voire impossible) pour le graphiste. En effet, un langage de programmation requiert en général des connaissances et aptitudes particulières pour être compris (alors qu'un langage XML, plus simple et plus intuitif, peut convenir à la création d'UI). Dès lors, les développeurs doivent réaliser eux-mêmes l'environnement graphique qui permettra d'interagir avec l'utilisateur. De par les exigences pointues des designers et le caractère changeant de la tâche, les programmeurs consomment un temps qui leur est précieux ! La conception des interfaces affecte ainsi considérablement leur travail en les empêchant de mener à bien les autres tâches qui leurs sont assignées.

Mission Critical (alias MC), une société belge de développement logiciel, s'est interrogée sur la situation. Cette entreprise spécialisée en programmation déclarative travaille avec le langage Mercury que nous verrons au chapitre 4. Les expériences que MC a tiré de ses projets passés l'a amenée à se poser des questions en la matière : comment permettre au designer d'accomplir son travail indépendamment du développeur ? Comment simplifier la génération graphique tout en conservant le dynamisme ? Peut-on trouver une solution qui permette de formuler les besoins par une approche déclarative ? *Mission Critical* souhaitait créer un langage de template (de syntaxe XML) proche de la sémantique Mercury qui permette de générer dynamiquement du XML. Au chapitre "*5. MSP*" nous verrons que *Mercury Server Page* présente une solution générique qui répond à ces questions. Le chapitre "*4. Introduction aux technologies et langages utilisés*" précède cette présentation et amène les notions nécessaires à la compréhension de MSP. Enfin, le chapitre "*6. Conclusion*" clôturera cette rédaction.

Chapitre 2

XML en bref

2.1 Présentation générale

XML (eXtensible Markup Language) est un langage de balisage qui offre une "syntaxe standard pour décrire des structures arborescentes sous forme textuelle" [16]. Un document XML est ainsi constitué d'un ensemble de tags[1] qui entourent des chaînes de caractères. Au départ créé pour combler les faiblesses du HTML, deux changements essentiels ont été apportés : XML ne prédéfinit pas de balises et il est plus strict que HTML [23]. Le langage XML, développé par un groupe du W3C en 1996, a connu une propagation très rapide. Il est aujourd'hui mondialement répandu, reconnu et utilisé ; c'est un standard qui sert de référence pour de nombreuses applications. XML trouve son utilité tant pour représenter de l'information que pour permettre l'interopérabilité entre systèmes hétérogènes (besoin qui se fait fortement ressentir avec l'évolution croissante des webservices). Sa syntaxe est claire, simple et bien définie. L'opportunité de s'en servir comme format standard a rapidement été saisie étant donné sa portabilité au travers des divers systèmes et plateformes.

Bien que les utilisations qu'on en fait soient très larges, il est important de garder en vue ce qu'est XML intrinsèquement. "Il procure une syntaxe aux documents structurés mais n'impose aucune contrainte sémantique sur leur siginfication" [13]. En effet, ce n'est qu'un langage standardisé qui permet de structurer des données. A la base aucun sens, aucun mécanisme, aucun dynamisme n'y est associé. XML en soi est purement statique, il ne fait rien. Ce n'est donc certainement pas un langage de programmation, un protocole réseau ou une base de données [18].

Pour consulter un document XML il n'est pas nécessaire d'avoir recours à un éditeur spécifique, tout programme capable de lire du texte peut ouvrir un tel ficher. Il existe néanmoins des programmes spécialisés, appelés *parseurs*, qui

[1] synonyme de balise

permettent de parcourir un document XML et d'en extraire le contenu.

EXEMPLE introductif :

```
<message to="you@yourAddress.com"
from="me@myAddress.com"
             subject="XML Is Really Cool">
   <text>
     How many ways is XML cool? Let me count the ways...
   </text>
</message>
```

Exemple recopié d'un cours de Jacques Lemordant [22].

- *you@yourAddress.com* est la valeur de l'attribut "to".
- La balise "</message>" représente la fin de l'élément "message".

Comme le montre l'exemple ci-dessus, une balise XML a la forme "`<Mon_tag>`". L'*élément* est l'unité de base, il est composé d'une balise ouvrante et d'une balise fermante (du même nom[2]) qui le délimitent. Il peut posséder des *attributs* dont la valeur doit être stipulée entre ses guillemets, derrière le caractère "=". Toute balise ouverte doit être refermée (via "`</Mon_tag>`" si la balise ouvrante était "`<Mon_tag>`") et tout document XML doit avoir un élément racine qui englobe tous les autres.

```
<Racine>
   <Element1>
   </Element1>
   <Element2>
   </Element2>
   ...
</Racine>
```

La description de la syntaxe XML par le W3C est disponible en ligne. L'objet ici n'étant pas d'enseigner la syntaxe du langage, des informations complémentaires à ce sujet pourront donc être trouvées sur le site [30].

[2]il est important de noter que XML est sensible à la casse

2.2 conformité

Le respect des règles par un document XML se teste à deux niveaux : on marque une distinction entre sa conformité vis-à-vis de la syntaxe et sa conformité vis-à-vis d'une DTD (Document Type Definition) ou d'un schéma[3]. Une DTD offre la possibilité de spécifier une syntaxe sur base de laquelle un document XML peut être vérifié. Elle exprime rigoureusement ce que l'on peut retrouver dans le document et sous quelles contraintes. "La validité opère selon le principe que tout ce qui n'est pas autorisé est interdit" [18].

EXEMPLE de DTD (annuaire.dtd)[4] :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT annnuaire (personne*)>
<!ELEMENT personne (nom,prenom,email+)>
<!ATTLIST personne type (étudiant | professeur | chanteur | musicien)
"étudiant">
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

L'exemple de DTD donné précédemment exprime une syntaxe où un élément "annuaire" peut contenir une ou plusieurs personnes. Pour qu'un document XML soit valide vis-à-vis de cette DTD, il faut que chaque personne ait un nom, un prénom et au moins une adresse e-mail. Une personne possède un attribut "type" qui peut prendre la valeur "étudiant", "professeur", "chanteur" ou "musicien" ("étudiant" étant la valeur par défaut). Les éléments "nom", "prenom" et "email" sont des chaînes de caractères.

Il n'est pas impératif pour un document XML de suivre la syntaxe définie par un schéma. conforme au fichier DTD, le document est dit "valide". En cas de respect de la syntaxe XML il aura tout de même droit au titre de document "bien formé". Un document peut ainsi être *bien formé* mais *invalide*. Il est évidemment indispensable de se conformer aux règles syntaxiques émanant de la norme du W3C qui représentent la base même du langage, les processeurs XML rejeteront les documents qui ne sont pas *bien formés*.

[3] un schéma XML représente conceptuellement la même chose qu'une DTD mais permet des définitions plus détaillées

[4] http ://www.toutestfacile.com/xml/cours/printables/XMLFacile.com-dtd.php

Un document **valide** doit satisfaire quatre critères[17] :

1. Etre bien formé.
2. Comporter une déclaration de type de document[5].
3. Son élément racine doit correspondre à celui spécifié par la déclaration du type de document.
4. Satisfaire toutes les contraintes indiquées par la DTD spécifiée par la déclaration de type de document.

EXEMPLE de document XML valide pour *annuaire.dtd*[6] :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE annuaire SYSTEM "annuaire.dtd">
<annuaire>
     <personne type="étudiant">
         <nom>HEUTE</nom>
         <prenom>Thomas</prenom>
         <email>webmaster@xmlfacile.com</email>
      </personne>
     <personne type="chanteur">
         <nom>CANTAT</nom>
         <prenom>Bertrand</prenom>
         <email>noir@desir.fr</email>
     </personne>
</annuaire>
```

La ligne `<!DOCTYPE annuaire SYSTEM "annuaire.dtd">` est la déclaration de type de document. Cela signifie que le document ci-dessus doit respecter la syntaxe définie par *annuaire.dtd* pour être valide et que son élément racine doit être "annuaire".

[5] la **déclaration** de type de document et la **définition** de type de document sont deux choses différentes. La déclaration est la référence dans le document XML qui stipule à quelle DTD il doit se conformer

[6] aussi issu de http ://www.toutestfacile.com/xml/cours/printables/XMLFacile.com-dtd.php

2.3 Les espaces de noms

Le succès de XML s'étend au-delà des frontières d'Internet et dans un tas de domaines. Il en résulte une forte utilisation du langage, et ce dans une multitude de contextes différents. Il en résulte un problème de compréhension et de compatibilité : un document X peut employer un vocabulaire identique à un document Y sans vouloir désigner la même chose. Si l'on tente de mêler l'information de X et Y ou tout simplement si l'on parcourt un document sans connaître le domaine auquel il se rapporte, l'information devient ambigüe (voire incohérente si on l'interprète mal). Cela s'explique par le fait que, dans des contextes différents, les mots peuvent avoir des significations différentes. Les *espaces de noms* répondent à ce problème en associant à leur contexte les divers termes employés au sein d'un document XML. De cette manière, l'ambiguité est levée et les concepts sont clairement identifiés.

Les espaces de noms sont implantés dans un fichier XML à l'aide de préfixes que l'on associe à une URI[7]. L'URI sert d'identifiant et est liée à un *namespace*[8] par une déclaration via le préfixe **xmlns**. Par définition xmlns est lui-même associé à l' de nom `"http://www.w3.org/2000/xmlns/"` mais cela ne dois pas être déclaré[32].

EXEMPLE (issu du site du W3C [32]) :

```
<x xmlns:edi='http://ecommerce.example.org/schema'>
   <!-- the "edi" prefix is bound to http://ecommerce.example.org/schema
         for the "x" element and contents -->
</x>
```

2.4 Le traitement XML

En ce qui concerne la conception de programmes traitant des documents XML on distingue trois approches possibles [16] :

La première est de partir d'un langage de programmation classique (C, Java, ...) qui sera couplé avec des librairies spécialisées. Ces dernières complètent le langage en apportant des fonctionnalités de traitement XML (parsing, parcours de l'arbre, récupération des valeurs,...). Deux interfaces standard utilisées

[7] *Uniform Resource Identifier* est une chaîne de caractère qui identifie une ressource Web physique ou abstraite. Elle peut être de type *locator* (URL) ou de type *name* (URN).[3]

[8] *Namespace* est l'équivalent d'*espace de nom* en anglais

pour ces manipulations sont DOM[9] et SAX[10]. Ce type d'approche a pour avantage d'éviter la formation des programmeurs à un nouveau langage. L'inconvénient qui en découle est que ce langage n'est en principe pas conçu pour le traitement XML. Sa syntaxe inadéquate nuit à la clarté du code.

Le deuxième niveau de traitement consiste à utiliser un langage générique spécifiquement étendu pour le traitement XML[11]. Dans ce cas, le langage est plus adapté à l'usage qu'on en fait et la détection des erreurs s'en retrouve également facilitée.

Enfin, la troisième approche se fonde sur l'emploi d'un langage spécifique à XML (comme par exemple XSLT, voir section 3.2). Dans ce niveau de traitement les solutions sont expressément conçues pour être appliquées à des cas XML. Leur syntaxe est donc parfaitement adaptée, entraînant ainsi une concision et une lisibilité nettement meilleures en comparaison avec les deux autres approches. Les constructions de haut niveau améliorent la productivité des programmeurs, et une gestion des erreurs liées aux manipulations XML est assurée. L'inconvénient majeur réside dans le fait que si ces langages sont spécifiques, ils demandent logiquement un apprentissage spécifique.

[9] *Document Object Model* parse le document XML, le transforme en arbre et offre une interface qui permet de manipuler les éléments de cet arbre

[10] *Simple API for XML* est constitué d'un parseur événementiel : lorsqu'il rencontre (par exemple) une ouverture de balise, un noeud de texte, la fin du document... il déclenche un évènement (en temps réel). Un évènement se traduit par l'appel d'une méthode qui peut être implémantée par le programmeur pour y associer une action

[11] comme XJ (basé sur java) ou HaXML (qui repose sur Haskell)

Chapitre 3

Quelques solutions existantes

XML est statique en soi mais bien avant MSP des solutions ont été élaborées pour générer du contenu avec dynamisme. Parmi celles-ci notons la famille des "*server pages*" (ASP, JSP, HSP,...), CGI et XSLT auxquels nous allons porter notre attention dans ce chapitre. Le but ici n'est évidemment pas de donner une description complète de chacune des technologies mais plutôt de les introduire (sans rentrer excessivement dans les détails) pour parcourir le domaine de la génération dynamique de XML.

3.1 Le Web dynamique (introduction par Java Server Page)

La génération dynamique de XML est principalement présente dans le contexte du Web et JSP est une des technologies spécialement conçues pour répondre à ce problème. Intervenant dans une architecture client-serveur, elle permet de générer dynamiquement le résultat renvoyé par le serveur au client (qui l'interroge à l'aide d'un navigateur Internet). Nous aborderons le sujet dynamique en parcourant JSP à la suite de quoi nous décrirons quelques alternatives à la solution proposée par Java. Avant d'entamer le sujet des Java Server Pages, commençons par considérer ce qu'est une *servlet* et comment cela fonctionne.

3.1.1 Les servlets

Une servlet est un composant Java implanté du côté du serveur. Ce dernier fait appel aux servlets pour pouvoir donner suite aux requêtes émanant du client. La plupart du temps, dans le contexte du Web, les requêtes du client concernent des pages HTML entièrement statiques et dans ce cas, le serveur joue un simple rôle de bibliothécaire [19] : il va chercher dans sa mémoire la page qui lui a été demandée et la renvoie.

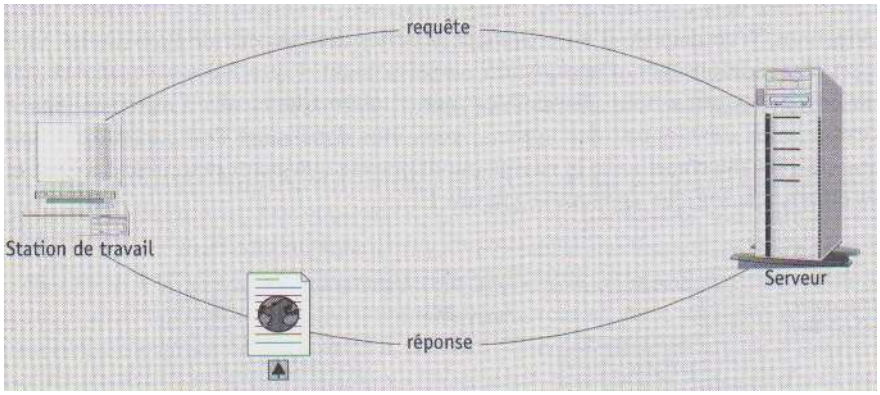


FIG. 3.1 – Architecture client-serveur [19]

Bien évidemment une servlet peut faire bien plus que cela puisqu'elle est constituée de code Java qui lui permet d'invoquer des méthodes, de traiter des données ou de manière générale, de calculer et fabriquer un résultat. Ainsi, il est possible de répondre dynamiquement à une requête à l'aide d'une servlet.

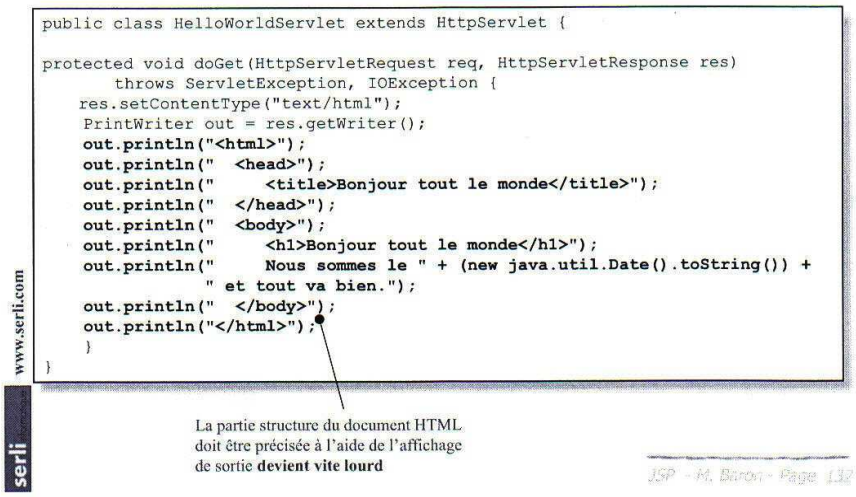


FIG. 3.2 – Génération de HTML avec une servlet

Néanmoins, comme l'illustre la figure 3.2 (issue d'un slide show de *M. Baron* [5]) la production de documents HTML au sein d'une servlet Java est fastidieuse. De plus, un designer qui devrait créer une interface Web se retrouverait

bloqué face à la servlet car il ne connaît pas la programmation Java.

3.1.2 L'intervention de JSP

Les JSP, qui tout comme les servlets permettent de créer du contenu Web avec dynamisme, apportent une aide précieuse dans ces circonstances. Un fichier JSP n'est pas construit comme un programme qui intègre dans ses lignes des bribes de HTML, mais plutôt comme un fichier HTML[1] entrecoupé de passages en code Java. Comme pour les autres membres de la famile "*server pages*" les instructions de programmation sont délimitées par des balises `<% incursion de java %>` qui isolent le code dans des blocs distincts du reste du contenu. Le code est exécuté au vol et remplacé au final par le résultat de son exécution [24].

L'équivalent de la figure 3.2 en JSP donne ceci[2] :

```
<html>
  <head>
    <title>Bonjour tout le monde</title>
  </head>
  <body>
    <h1>Bonjour tout le monde</h1>
    Nous sommes le <%= new java.util.Date().toString() %> et tout va bien.
  </body>
</html>
```

La raison maîtresse qui nous a poussé à introduire JSP par les servlets est qu'en réalité, un fichier JSP est amené à être compilé en une servlet. La figure 3.3 (tirée de l'ouvrage de *J. Weaver, K. Mukhar* et *J. Crume* [19]) explicite le fonctionnement : la JSP est compilée lors de la **première utilisation**. Les demandes suivantes seront assurées directement par la servlet générée, garantissant ainsi une meilleure efficacité.

L'intervention d'une JSP reste transparente pour le client, le résultat qu'il reçoit est dépourvu de tout code Java. Les Java Server Pages seront converties en servlets sur le serveur, qui les exécutera par la même occasion et renverra uniquement du HTML.

[1] Une JSP peut aussi contenir du XML

[2] Cette traduction de la figure 3.2 en JSP (recopiée de la même source) illustre clairement l'intérêt de recourir aux Java Server Pages

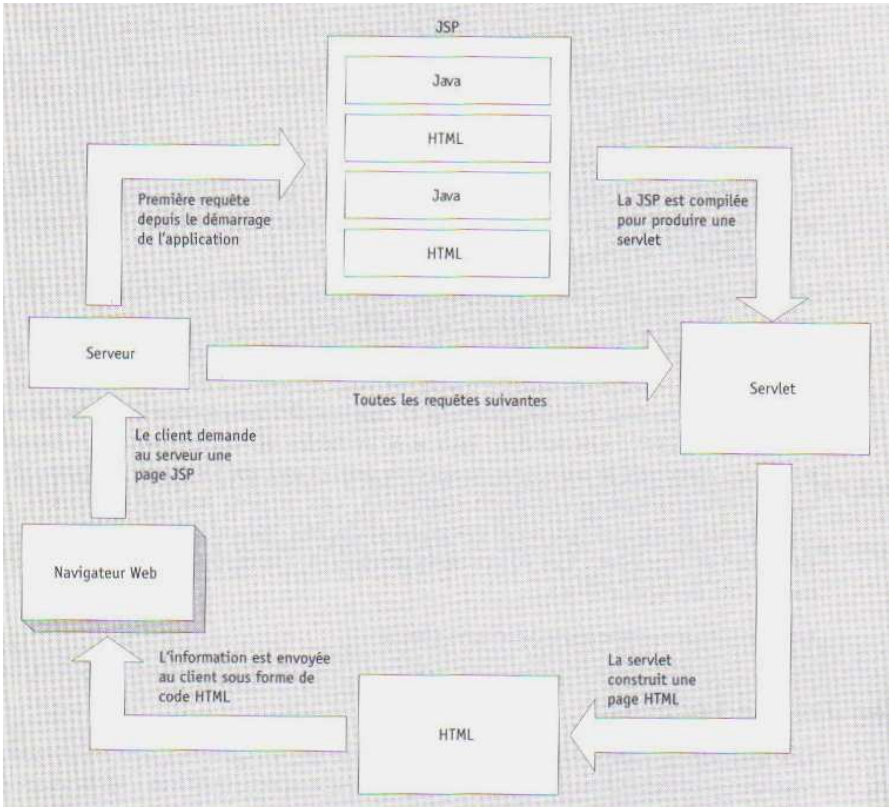


FIG. 3.3 – De la JSP à la servlet

Contrairement aux applets[3] ou au Javascript[4] qui s'exécutent du côté du client, avec JSP le travail est réalisé du côté du serveur. D'autres technologies fonctionnent de cette manière, nous le verrons à la section 3.1.4.

[3] "Les applets sont différentes des applications dans la mesure où il s'agit de petites applications graphiques destinées à s'exécuter dans un navigateur Internet" [1]

[4] "JavaScript est un langage de programmation non compilé, orienté objet, principalement utilisé dans les pages Web" [3]

Les technologies côté serveur permettent [10] :

- L'exécution de programmes écrits dans des langages qui ne sont pas supportés par le navigateur du client.
- La production de pages Web dynamiques sans devoir recourir à des fonctions spécifiques du côté du client.
- De disposer de données que le client ne possède pas.
- La limitation du poids de chargement à une simple page HTML.
- Une meilleure sécurité car le code est invisible pour le client.

Les conséquences se font par contre ressentir au niveau du serveur : il doit maintenant réaliser le travail qui, dans l'autre scénario, était assuré par le client. Chaque client consommait ses propres ressources pour sa propre utilisation, mais dans ce cas ci le serveur réalise les calculs pour tous ! Plus les clients sont nombreux et plus le serveur est sollicité... il est donc nécessaire de prévoir des capacités de calcul pour supporter la charge.

Les technologies côté serveur sont couramment utilisées car les points positifs qu'elles apportent prennent souvent le dessus face aux inconvénients qu'elles entraînent, mais le choix est laissé à l'appréciation de chacun. L'intérêt principal des technologies côté client est qu'elles permettent de créer des pages Web plus interactives. Rien n'empêche toutefois de générer dynamiquement avec JSP des pages qui contiennent, par exemple, des passages de Javascript. Dans ce cas les technologies client et serveur sont utilisées conjointement.

3.1.3 Balises personnalisées

(*Cette sous-section est entièrement inspirée du livre "J2EE 1.4" [19]*)

Comme cela a été mentionné, JSP permet de séparer le contenu HTML (ou XML) du code Java qui lui procure son dynamisme. On sépare ainsi la représentation de la programmation (même si tous deux occupent le même fichier). Il reste néanmoins que plus l'application est complexe, plus la longueur des blocs de code présents est conséquente. Finalement, le graphiste pourrait de nouveau se perdre si l'occurence de ces blocs devenait trop usuelle et surtout si leur taille tendait à s'allonger de manière disproportionnée par rapport à la quantité de HTML présent. Pour remédier à ce problème, il est possible de remplacer le code Java par des balises personnalisées dans les pages JSP. Avec cette technique, on rend au HTML une place prépondérante.

Les actions personnalisées prennent donc la forme de balises courantes. On les identifie par un préfixe et un nom : `<préfixe:nom />`. Le préfixe indique la librairie à laquelle se rapporte l'action (pour éviter les conflits entre les différentes librairies) et le nom identifie l'action elle-même. Un document XML appelé TLD (Tag Library Descriptor) permet de déterminer quels gestionnaires

de balises[5] sont disponibles dans une bibliothèque. Ainsi, lorqu'une balise personnalisée est trouvée dans une JSP, son préfixe identifie la librairie concernée et le TLD indique quelle classe (ou gestionnaire de balises) traite l'action en question. Lors de l'exécution, une instance de cette classe déterminera le comportement de l'action.

Avant toute utilisation de tags personnalisés, il faut importer la librairie à laquelle ils appartiennent. Ceci est réalisé par la présence de la directive :

```
<% taglib uri="URI_bibliothèque" prefix="préfixe_des_balises" %>
```

Après l'occurence de cette ligne, les balises personnalisées débutant par le préfixe spécifié dans l'attribut "prefix" seront associées à des actions appartenant à la librairie mentionnée dans l'attribut "uri".

A ce sujet, il existe une librairie standard de balises nommée JSTL (Java server pages Standard Tag Library), qui reprend de nombreuses tâches fondamentales. En effet, il a été constaté que beaucoup de développeurs s'évertuaient à créer des bibliothèques personnalisées offrant des fonctionnalités souvent fort proches. La JSTL a alors standardisé les implémentations d'une multitude de besoins fréquemment rencontrés. Dans de nombreux cas, il est désormais possible de se procurer une implémentation satisfaisant à ces besoins sans devoir créer soi-même sa propre bibliothèque personnalisée.

[5] un gestionnaire de balises est une classe Java implémentée pour décrire quel comportement adopter lorsque l'on rencontre une action personnalisée.

EXEMPLE DE BALISE PERSONNALISEE (issu de [6])

Une classe Java qui écrit "hello world" dans une page :

```
package monpackage;
 ...
 public class HelloTag extends TagSupport {
     public int doStartTag() throws JspException {
          try {
            pageContext.getOut().println("Hello World !");
         } catch (IOException e) {
           throw new JspException ("I/O Error", e);
         }
         return SKIP_BODY;
    }
 }
```

Un fichier TLD qui lie l'action "hellotag" à la classe ci-dessus :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib ...>
   <tlib-version>1.0</tlib-version>
   <jsp-version>1.2</jsp-version>
   <description>
         Bibliothèque de taglibs
   </description>
   <tag>
   <name>hellotag</name>
   <tag-class>monpackage.HelloTag</tag-class>
   <description>
         Tag qui affiche bonjour
   </description>
   <body-content>empty</body-content>
   </tag>
</taglib>
```

Un fichier "web.xml" qui fait le lien entre la JSP et le TLD :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
<display-name>Permet de gérer des Tags personnalisés</display-name>
<taglib>
     <taglib-uri>monTag</taglib-uri>
     <taglib-location>/WEB-INF/tld/montaglib.tld</taglib-location>
</taglib>
</web-app>
```

La JSP qui comporte le nouveau tag :

```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:hellotag /> Tout le monde
```

Et enfin le résultat :

FIG. 3.4 – Résultat de TagLib

Dans la JSP on importe tout d'abord la bibliothèque `monTag` dont l'URI réelle est en fait définie dans "web.xml" par `"/WEB-INF/tld/montaglib.tld"`. On associe à cette bibliothèque le préfixe `montagamoi` (première ligne de la JSP). Ce préfixe est désormais lié à `"/WEB-INF/tld/montaglib.tld"`, le *Tag Library Descriptor* qui décrit les actions disponibles[6]. Avec tout cela, `<montagamoi:hellotag/>` peut enfin être remplacé par "Hello World !".

[6]Dans le cas présent, il n'existe que l'action `hellotag` implémentée par la classe *HelloTag*

3.1.4 Technologies similaires

JSP n'est pas la seule technologie existante en matière de génération dynamique de pages Web, loin de là. D'autres solutions permettent de tenir compte du contexte environnant ou de la survenance d'évènements pour adapter la réponse adressée au client. Pour ce dernier, rien ne change : tant que les opérations à l'origine du dynamisme sont entièrement assurées par le serveur, le client effectue une simple demande de page et ne calcule rien. Si cette page nécessite un processing pour tenir compte de l'aspect dynamique, c'est le serveur qui effectuera la tâche et génerera le HTML résultant. Les technologies côté serveur suivent toutes la même démarche (comme le montre la figure 3.5, tirée du livre *Initiation à ASP 3.0* [10]).

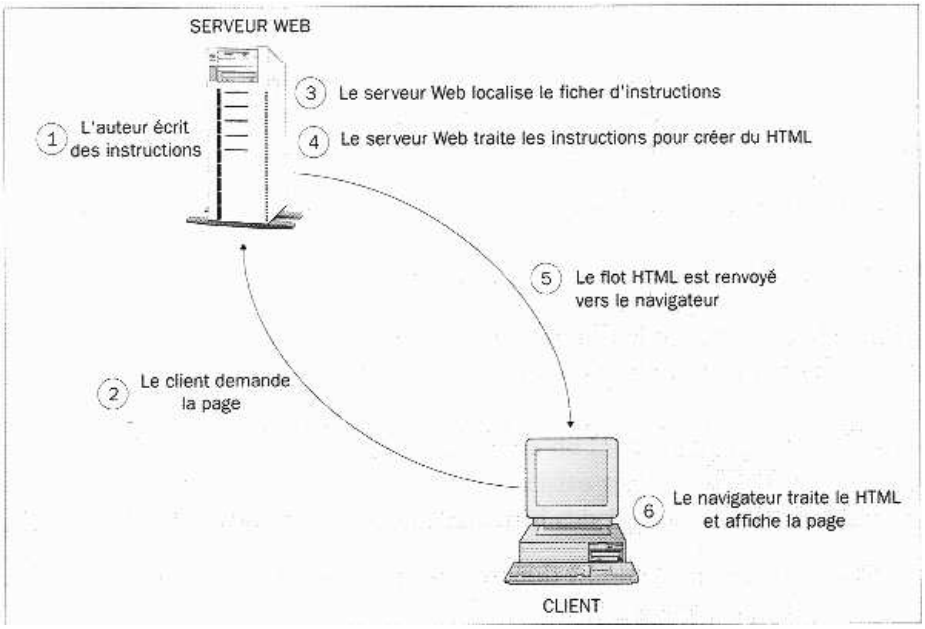


FIG. 3.5 – Technologie côté serveur

Active Server Page (ASP)

ASP est l'une des solutions concurrentes à JSP. Cette technologie se trouve liée à *Microsoft* et donc à IIS[7] en ce qui concerne les serveurs. Active Server Page n'est pas un langage de programmation en soi, il se définit comme étant "une technologie qui autorise la construction programmatique de pages HTML juste avant de les envoyer au navigateur" [10]. Pour ce faire, il recourt à des scripts qui assurent son dynamisme. Comme pour JSP, les scripts sont délimités du reste de la page par des balises `<% contenu du script %>` et le principe de fonctionnement est exactement le même. Deux langages de scripts sont principalement supportés, à savoir Javascript et VBScript. Un point qui les différencie tout de même est la portabilité : qui dit Java dit indépendance du système utilisé. Cela permet aux Java Server Pages d'être implémentés sur différents serveurs, tandis que ASP restera cadenassé au monde Microsoft.

Common Gateway Interface (CGI)

Cette solution (apparue quelques mois avant ASP) réalise l'interfaçage entre le serveur Web et des **programmes externes** qui insufflent le dynamisme dans la génération des pages. CGI ne fait que transmettre les données du serveur aux programmes (où elles seront traitées) puis retourner le résultat au serveur. N'importe quel langage de programmation peut satisfaire à la rédaction d'un script CGI même si la plupart du temps, c'est le langage *Perl*[8] qui est employé [3]. Cependant, la technologie *Common Gateway Interface* implique l'utilisation d'une syntaxe particulière qui rend la plupart des langages de programmation peu adaptés au problème. Il est préférable de choisir un langage capable de communiquer facilement avec d'autres logiciels et qui possède des fonctionnalités puissantes pour manipuler les chaînes de caractères [10]. Un point faible de cette technologie réside dans le fait qu'elle cause souvent des failles en matière de sécurité. Elle constitue un moyen de toucher le système d'exploitation de la machine sur laquelle tourne le seveur [11]. CGI reste tout de même la technologie la plus utilisée dans le domaine de la génération dynamique de pages Web.

[7] "Internet Information Services, communément appelé IIS, est le logiciel serveur Web de la plateforme Windows NT." [3]

[8] "Perl (Practical Extraction and Report Language) est un langage de programmation dérivé des scripts shell [...]. Il s'agit d'un langage interprété dont l'avantage principal est d'être très adapté à la manipulation de chaînes de caractères." [1]

Personal Home Page (PHP)

PHP est un langage de scripts libre, gratuit et multi-plateformes. La plupart du temps on le retrouve sur un serveur HTTP pour générer des pages Web dynamiques. Typiquement, suite à la demande d'un client, le serveur interprète les instructions PHP qui génèrent du HTML et le résultat est renvoyé au browser. Cependant, ce langage de script peut aussi être interprété localement via l'exécution de programmes en ligne de commande [3]. PHP n'est pas un simple langage de script, il est compatible avec la majorité des bases de données et des librairies externes qui élargissent l'éventail de ses capacités (il peut par exemple analyser du XML [10]). Sa syntaxe est assez proche du *Perl* ou du *C* mais contrairement à ces langages il est aussi prévu pour être injecté dans du code HTML par un système de balises `<?php ... ?>`. La possibilité d'utiliser PHP en tant que script CGI existe également.

Haskell Server Page (HSP)

Basé sur le langage de programmation Haskell[9], HSP appartient lui aussi à la famille des *server pages*. Contrairement aux langages armés d'un typage faible (comme Javascript, Perl,...) HSP s'entoure d'une technologie assurant une meilleure fiablilité dans la survenance de crash et dans la garantie d'obtenir des réponses correctes [14]. Des outils tels que *HaskellDB* (pour l'accès aux données), *HaskellScript* (pour la rédaction de scripts dans l'optique DHTML[10]) ou *HaskellCGI* (librairie conçue pour la création de scripts CGI) peuvent être couplés à Haskell Server Page et élargir ainsi l'étendue de ses possibilités. Une caractéristique intéressante du HSP est que les fragments XML ont le même statut qu'une expression Haskell [8] :

```
helloWorld = <p>Hello World!</p>
```

Etant donné que XML a été rajouté dans les types d'expressions, les éléments XML peuvent être insérés dans des listes, passés comme des arguments ou encore retournés par une fonction. Un autre point intéressant s'observe dans la possibilité d'appliquer le *pattern matching* sur des fragments XML. Cela permet, par exemple, de transformer un document XML en un autre (illustration tirée du papier [14]).

[9] Haskell est un langage de programmation fonctionnelle créé en 1985 par le mathématicien et logicien Haskell Brooks Curry[3]

[10] Le *Dynamic HyperText Markup Language* n'est pas régi par une norme, c'est simplement du HTML que l'on a rendu interactif en y insérant des mécanismes qui créent du dynamisme (comme JavaScript)

SOURCE XML

```
<MSG>
  <FROM>erik@meijcrosft.com</FROM>
  <RCPT>
    <TO>billg@microsoft.com</TO><TO>steveb@microsoft.com</TO>
  </RCPT>
  <SUBJECT>HSP</SUBJECT>
  <BODY>
    <P>HSP is cool, isn't it?</P>
  </BODY>
</MSG>
```

RESULTAT

```
From: erik@cs.uu.nl
To: billg@microsoft.com, steveb@microsoft.com
Subject: HSP
HSP is cool, isn't it?
```

FONCTION DE CONVERSION (nommée "toRFC822")

```
toRFC822 :: MSG -> RFC822
toRFC822 =
\<MSG>
  <FROM><% [from] %></FROM>
  <RCPT><% rcpts %></RCPT>
  <SUBJECT><% [subject] %></SUBJECT>
  <BODY><% paras %></BODY>
</MSG> ->
   concat
     [ "From: " ++ from, crlf
     , "To: " ++ intersperse "," (stripTOs rcpts), crlf
     , "Subject: " ++ subject, crlf, crlf
     , intersperse crlf (stripPs paras), crlf
     ]
```

La fonction *stripTOs* extrait la liste des destinataires et *stripPs* récupère le contenu des paragraphes.

Typiquement Haskell Server Page a été conçu pour tourner sur un serveur HTTP et renvoyer du contenu HTML (ou XML). Comme les autres "*server pages*", une page HSP peut inclure des scripts dont le début et la fin sont marqués dans un tag.

3.2 XSLT

XML tend à exprimer des structures de données (à leur fournir un balisage pour les organiser clairement) mais un document XML ne contient en général pas d'information quant à la représentation de ces données. Bien qu'il ne se limite pas à cet usage, XSLT (*eXtensible Style Language Transformation*, défini par le W3C [29]) offre typiquement une solution à ce problème en permettant de spécifier des règles de transformation à appliquer sur un document XML. De cette manière on peut concevoir une *feuille de style*[11] qui aura pour effet de rajouter des informations de mise en page à des données XML.

3.2.1 Fonctionnement

Comme l'exprime le schéma de la figure 3.6 (tiré de l'ouvrage "XML by example" [23]), XSLT prend un document XML en entrée, s'appuie sur les règles définies dans une feuille de style pour le transformer, et produit un document XML en sortie (résultant de la transformation de l'entrée).

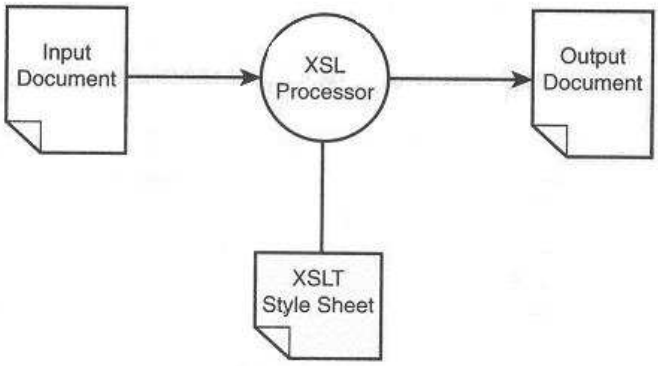


FIG. 3.6 – Transformation XSLT

[11] Une feuille de style est un document constitué d'un ensemble de règles permettant de mettre en forme un autre document[3]

Au vu de cette définition, il ressort en effet que cela ne se limite pas à de la mise en page de données, on pourrait tout aussi bien restructurer le contenu du document dans un arrangement différent. Cela revient simplement à modifier la structure de l'arbre, comme dans la figure 3.7 (provenant d'un cours de Pierre Genevès [16]).

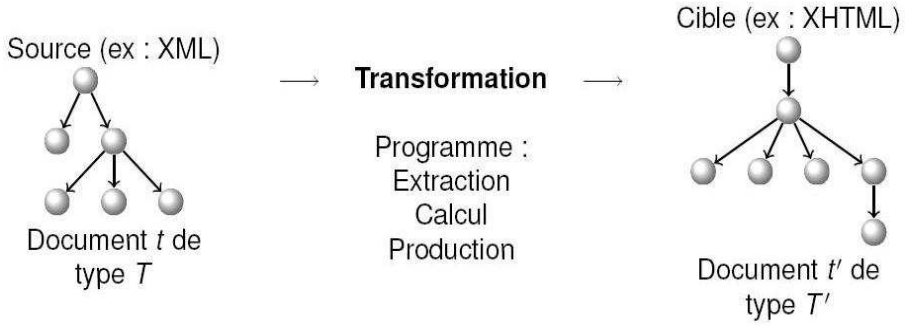


FIG. 3.7 – Transformation d'arbre

La place de XSLT dans ce mémoire pourrait être discutée. Contrairement aux autres solutions qui **produisent** du XML (sans document entrant), XSLT opère un traitement sur des données préexistantes. De plus, toutes les transformations d'un même document sur base d'une même feuille de style produira toujours le même résultat. Le dynamisme n'est donc pas présent au départ.

Par contre, on peut insérer du dynamisme dans un résultat XSLT en générant du DHTML [2]. Dans ce cas l'aspect dynamique est transféré au résultat produit et non à XSLT lui-même. Selon cette référence [2], une deuxième approche permettrait d'aller plus loin en donnant la possibilité de modifier (via JavaScript ou DOM) la feuille de style elle-même.

Dans l'introduction à XML (chapitre 1), il a été mentionné que ce langage est un succès compte tenu de sa neutralité et de sa portabilité au travers de systèmes hétérogènes (diversité tant matérielle que logicielle). Son utilisation convient donc pour structurer et/ou véhiculer de l'information (à titre d'exemple, du contenu Web qui doit pouvoir être interprété par un PC, un PDA, ou encore un GSM). En couplant XML avec XSLT, on peut parfaitement scinder les données de leur représentation ou de leur format final : les données sont contenues dans un document XML source et les informations de conversion vers une représentation spécifique sont conservées chacune dans une feuille de style différente. Ainsi, à partir d'une même source on peut générer différents résultats (conserver une même sémantique dans des syntaxes différentes).

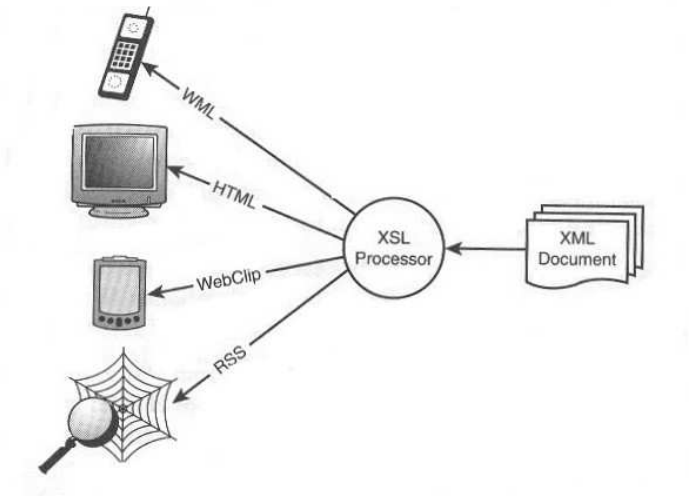


FIG. 3.8 – La portabilité de XML et XSLT

3.2.2 Les feuilles de style

Une feuille de style XSL est un document XML bien formé. Les éléments XSLT appartiennent tous à l'espace de nom

`http://www.w3.org/1999/XSL/Transform`

que l'on associe généralement au préfixe "xsl". L'élément racine d'une feuille XSL doit être soit "`<xsl:stylesheet>`" soit "`<xsl:transform>`" (qui sont en fait des synonymes).

Le résultat de la transformation d'un document est obtenu par application des règles définies dans la feuille de style. Si sa racine n'a aucun enfant, on se retrouve face au document XSLT minimal dont l'action ôtera toutes les balises du document d'entrée (il ne restera donc que du texte). Un élément "`<xsl:template>`" représente une règle et est pouvru d'un attribut *match* déterminant quels éléments du document d'entrée doivent être "matchés" par cette règle. Un template est en fait constitué de deux parties : la première lui permet de matcher les éléments à traiter (c'est ce qu'on appelle le "motif") et la seconde lui indique le comportement à adopter lorsqu'un matching survient (pour constituer une partie de l'arbre résultat) [29]. Un élément qui n'est pas sélectionné n'est pas pris en considération... il disparaît tout simplement.

EXEMPLE BASIQUE (tiré de [26])

Avec un document d'entrée :

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet type="text/xsl" href="message.xsl"?>

<doc>
  <message>Bonjour la Terre!</message>
  <message>Vénus aussi!</message>
</doc>
```

Et une feuille de style XSLT :

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet
         version="1.0"
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="doc">

   <html>
   <head>
          <title>Mon premier message</title>
   </head>
   <body>

          <xsl:apply-templates select="message"/>

   </body>
   </html>

</xsl:template>

<xsl:template match="message">

   <p><xsl:value-of select="."/></p>

</xsl:template>
</xsl:stylesheet>
```

On obtiendra le résultat :

```
<html>
<head>
       <title>Mon premier message</title>
</head>
<body>
       <p>Bonjour la Terre!</p>
       <p>Vénus aussi!</p>
</body>
</html>
```

Si l'on parcourt la feuille de style donnée en exemple, le premier élément intéressant rencontré est `xsl:template`. A l'intérieur de ce dernier, les éléments qui débutent par `"xsl:"` sont destinés à être interprétés tandis que les autres (qui ne sont pas des intructions XSLT) sont copiés tels quels dans le document produit. L'élément `<xsl:apply-templates select="message"/>` signifie que tous les éléments intitulés "message" qui sont enfants du noeud courant (à savoir "doc") doivent être traités ici. Cela renvoie à l'instruction `<xsl:value-of select="."/>` qui doit récupérer la valeur de l'élément (message) et l'insérer à l'endroit courant.

Nous ne verrons pas la syntaxe en détail mais des conditions supplémentaires peuvent être ajoutées pour bien cibler les éléments auxquels une règle doit être appliquée. La syntaxe qui permet de définir un chemin XML est appelée XPath [23]. Analogiquement, on peut la comparer au chemin d'accès d'un fichier.

Il peut arriver que plusieurs règles possèdent un motif qui corresponde à un même élément. Parmi elles, une seule devra être appliquée. Le premier critère de sélection est la *préséance d'import*. Cela signifie, lorsqu'une feuille de style en importe une autre, que la règle courante le remportera sur la règle importée. Sinon, la règle ayant la priorité la plus forte sera préférée. La priorité est déterminée par l'attribut "priority"[12] qui peut être associé à l'élément `<xsl:template>` [29].

[12] de valeur numérique

Chapitre 4

Introduction aux technologies et langages utilisés

Nous avons vu dans le chapitre précédent que diverses solutions existent pour générer du XML avec dynamisme. Avant de présenter MSP, il reste à introduire les technologies et langages nécessaires à sa compréhension. En effet, ce dernier sera balayé plus en détail (ce qui requiert des connaissances préalables plus affinées que pour les cas précédents). Dans cette quatrième partie nous parcourrons le langage de programmation *Mercury* (composante incontournable de MSP), la librairie *Mexpat* (utilisée pour le parsing XML) et le langage RDF (qui fait l'objet d'une instruction que nous verrons dans le chapitre suivant).

4.1 Mercury

Mercury est un langage de programmation logique et fonctionnel développé par l'Université de Melbourne[1]. La première version beta du langage accessible au public fut produite en 1995 [4]. Bénéficiant d'un algorithme d'exécution fortement optimisé, les programmes rédigés en Mercury jouissent d'une efficacité certaine pour un langage de cette catégorie[2].

4.1.1 Caractéristiques

Deux inconvénients majeurs accompagnent les langages de programmation logique : la lenteur d'exécution et le laxisme face aux erreurs (en comparaison avec les langages impératifs). En ce qui concerne Mercury, le premier argument

[1] Les notions élémentaires de programmation logique et de programmation fonctionnelle sont supposées connues par le lecteur

[2] Le benchmarking de Mercury a montré qu'il était presque deux fois plus rapide que le plus rapide des langages de programmation logique (Aquarius Prolog) et 20 à 36 fois plus rapide que certaines autres implémentations. Les détails sont disponibles à la page `http://www.cs.mu.oz.au/research/mercury/information/benchmarks.html`

est peu pertinent (comme expliqué au paragraphe précédent) et le deuxième est contré par un contrôle strict sur les types, les modes et le déterminisme[3] [33]. Une grande partie des erreurs est ainsi repérée avant l'exécution et le débogage est facilité.

Mercury est un excellent allié si l'on veut développer des programmes sans crash ni résultats erronés. Le fruit du travail des chercheurs australiens débouche sur un langage déclaratif de haut niveau conçu pour le développement de logiciels sûrs (même s'il sont d'une taille imposante) et efficace par des équipes de programmeurs [4]. Comme tout bon langage de programmation, Mercury "s'auto-compile" : le compilateur de Mercury est écrit en Mercury. Lors de la compilation le code sera converti en *C* (ce qui le rend relativement portable).

4.1.2 Le langage

En Mercury les variables commencent par des majuscules, tandis que les types, prédicats et les fonctions doivent avoir une minuscule en première lettre de leur nom.

Types

Le typage ressemble à ce que l'on rencontre dans les langages fonctionnels modernes[4] (comme Haskell). Une définition de type débute par "`:- type`".

```
:- type list(T) --->  [] ; [T | list(T)]
```

La ligne ci-dessus est un exemple de définition de type en Mercury. Elle est récursive car le membre de gauche "list(T)" est aussi présent dans le membre de droite. Si l'on traduit, cela signifie que *list(T)* est soit une liste vide (représentée par `[]`) soit une liste contenant des éléments de type T[5]. Dans ce dernier cas, on peut la décomposer en deux parties : la tête (un élément de type T) et le reste de la liste qui est elle-même une *list(T)* (d'où la récursivité de la définition).

[3] voir section 4.1.2

[4] voir le cours de *Programmation fonctionnelle et logique* [28]

[5] L'utilisation de "T" en majuscule ici représente n'importe quel type, mais tous les éléments de la liste devront être de même type. Si on instancie T avec un entier, on aura une liste d'entiers et on ne pourra pas y trouver par exemple d'éléments booléens

Prédicats

Pour créer un prédicat, il faut définir sa signature (où l'on déclare le type de ses arguments) et l'implémenter.

```
:- pred append(list(T), list(T), list(T)).
```

Une définition de signature de prédicat doit commencer par "`:- pred`" (dans le cas d'une fonction, ce sera "`:- func`") mais la ligne donnée ci-dessus ne suffit pas à elle seule, le compilateur Mercury exige que le mode (entrant/sortant) de chaque argument soit donné. Un prédicat peut être vu comme une relation tissée entre ses arguments où le mode détermine comment on va utiliser cette relation. Il faudra donc rajouter :

```
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
```

Ainsi, si l'on donne à *append* deux listes en entrée, il produira la concaténation de celles-ci en sortie ; si on lui donne juste une liste dans le troisième argument, il la scindera en deux listes qu'il renverra dans les deux premiers arguments. Le mode (in, in, out) implique aussi que l'on attend des deux premiers arguments qu'ils soient "ground", c'est à dire qu'ils soient instanciés par une valeur déterminée. Ces informations ne sont pourtant pas encore complètes pour MMC[6], il est obligatoire de spécifier le déterminisme. Cela consiste à préciser la certitude que l'appel d'un prédicat trouvera une réponse et, le cas échéant, le nombre de solutions qui peuvent être produites. Cette "utilisation particulière" du prédicat se fait en fonction du mode qui a été défini.

Ci-dessus, dans le premier cas le prédicat est "det" (pour déterministe). Cela signifie que si on associe effectivement deux listes à ses deux premiers arguments, il résultera de son appel une et une seule liste dans le troisième. Dans le cas du "multi" (pour multidéterministe), plusieurs résultats sont possibles ! Si on lui donne une liste pour troisième argument, il risque fortement d'y avoir plusieurs solutions. Par exemple si l'on prend "`append(A, B, [1,2,3,4,5])`", de multiples solutions pourraient être retournées :

`A=[1]` `B=[2,3,4,5]`, ou
`A=[1,2,3]` `B=[4,5]`, ou
etc...

Notons que "det" et "multi" ne sont pas les seules formes de déterminisme que l'on peut trouver, "semidet" par exemple désigne le cas où l'on peut pourrait obtenir soit une solution, soit aucune. Si un prédicat déterministe échoue, il déclenchera une erreur. Par contre un prédicat semi-déterministe couvre l'éventualité de l'échec, il ne déclenchera pas d'erreur s'il échoue.

[6] Melbourne Mercury Compiler

```
:- mode append(in, in, in) is semidet.
```

Ici le prédicat ne permet pas de trouver des solutions, son appel réussit si les deux premières listes sont une décomposition de la troisième et échoue sinon. Il est donc bien semi-déterministe. La notation permet d'abréger les signatures en regroupant le type des arguments, les modes et le déterminisme de la manière suivante :

```
:- pred date_anniversaire(int::in, int::in,
                          int::in, string::out) is det.
```

Cette déclaration de prédicat stipule que "date_anniversaire" possède 4 arguments (3 entiers et une chaîne de caractères). Les entiers sont utilisés en entrée par le prédicat (ils doivent être *ground* lors d'un appel à "date_anniversaire") tandis que la string a un mode "out" (ce sera donc une valeur retournée). L'occurence de "det" en fin de ligne précise que nous sommes dans un cas déterministe et que par conséquent, il y aura une et une seule solution. Suite à cela, il reste à implémenter le prédicat.

```
date_anniversaire(Jour, Mois, Annee, Date) :-

Date = int_to_string(Jour) ++ "-" ++ int_to_string(Mois) ++ "-"
       ++ int_to_string(Annee).
```

Les trois entiers précédemment cités correspondent respectivement à un jour, à un mois, et à une année que l'on va assembler pour constituer une date qui sera retournée sous forme de chaîne de caractères dans la variable "Date". Même sans connaitre Mercury, on comprend directement les opérations réalisées : les entiers sont convertis en string et concaténés (avec entre eux des guillemets) pour former une date (de type string) ayant la forme "Jour-Mois-Année".

En Mercury, les instructions de base sont l'appel à un prédicat et l'unification. Une instruction peut soit échouer, soit réussir (auquel cas plusieurs solutions sont parfois possibles, cf. multidéterminisme).

Conjonction et disjonction

Dans le cas d'une conjonction, les instructions sont séparées par une virgule. La conjonction `"(A, B)"` réussira si A et B réussissent. Elle peut réussir plus d'une fois si soit A ou B peut réussir plus d'une fois et que tout deux (A et B) peuvent réussir au moins une fois [15]. Si toutefois l'un d'eux échoue, la conjonction échouera fatalement.

Une disjonction sépare ses éléments par un point-virgule. Elle réussit au moins une fois si au moins un de ses bras réussit au moins une fois. Une disjonction échoue si chaque partie de la disjonction échoue. On parlera de *switch* si chaque partie de la disjonction teste une même variable liée, par exemple :

```
(
    L = [], empty(Out)
;
    L = [H|T], nonempty(H, T, Out)
)
```

La variable L est de type liste. Pour que le *switch* soit accepté par le compilateur, il doit couvrir tous les cas possibles de la liste et aucun cas ne peut satisfaire plusieurs bras à la fois.

Modules

Les programmes Mercury sont découpés en modules dans lesquels on retrouve les parties suivantes :

1. **:- module** nom_du_module.
2. **:- interface**.
3. **:- implementation**.

Un module est un fichier de la forme *nom_du_module.m* dont les premiers caractères visibles seront "`:- module nom_du_module.`" (avec un nom commençant par une minuscule puisque les majuscules sont réservées aux variables).

En-dessous vient la ligne marquant le début de la section interface (point 2 de l'énumération ci-dessus). On y trouvera les déclarations des types, des fonctions et des prédicats qui constituent l'interface du module (c'est-à-dire qui peuvent être utilisés par d'autres modules en cas d'exportation). Tout ce qui n'est pas déclaré dans cette partie ne pourra être utilisé que par le module lui-même. Si les déclarations nécessitent l'import de certaines librairies[7], c'est également ici qu'il faudra réaliser cette opération.

Enfin vient la partie implémentation (débutant par "`:- implementation.`"). Si des librairies autres que celles déclarées dans l'interface sont nécessaires, c'est ici qu'il faudra les importer. Bien évidemment, c'est dans la section implémentation que l'on va implémenter les fonctions et prédicats qui avaient été déclarés dans l'interface. On peut aussi écrire de nouvelles définitions (dont l'implémentation pourra cette fois directement être placée juste derrière vu que toutes deux se trouvent dans la même partie).

[7] L'import d'une librairie consiste à importer un module

EXEMPLE DE MODULE[7] :

```
:- module listeBasique.
:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is det.
:- implementation.
:- import_module list.

main(!IO) :- append([1,2,3,4],[5,6,7], MaListe).
```

Dans cet exemple nous appelons le prédicat *append* qui associera la variable *MaListe* à la liste `[1,2,3,4,5,6,7]`. Le prédicat *main/2* quant à lui doit être déclaré dans l'interface. Analogiquement, il correspond à la fonction *main* d'un programme *C* ou *Java* : lorsqu'un module a été compilé et qu'on l'exécute, c'est le prédicat *main* qui sera appelé[8]. Etant d'arité 2, il prend un argument IO (Input/Output) en entrée et un autre en sortie.

Gestion de l'IO

L'exemple classique "hello world !" [7] :

```
:- module hello.
:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is det.
:- implementation.
main(IOState_in, IOState_out) :-
   io.write_string("Hello, World!\n", IOState_in, IOState_out).
```

En Mercury la gestion des entrées/sorties est particulière. "Tout prédicat touchant à l'entrée/sortie doit avoir un argument IO en input déterminant l'état du monde au moment où le prédicat est appelé, et un argument IO en ouput déterminant l'état du monde après l'appel" (traduction non officielle [7]). Ainsi, si l'on avait effectué des impressions en chaîne sur la sortie standard, il aurait fallu écrire :

```
main(IOState_in, IOState_out) :-
    io.write_string("Hello, ", IOState_in, IOState_tmp1),
    io.write_string("World!", IOState_tmp1, IOState_tmp2),
    io.nl(IOState_tmp2, IOState_out).
```

[8] Un module ne contient pas obligatoirement de prédicat *main* mais sans ce dernier, il n'est pas exécutable

Dans la déclaration du *main*, deux modes particuliers ont été utilisés "`main(io::di, io::uo)`". Ces modes se justifient dans le cadre du système de changement d'états du monde auxquels Mercury recourt pour préserver son intégrité mathématique. Ils signifient respectivement *destructive input* et *unique output*.

4.2 Mexpat

Mexpat est un outil de traitement de documents XML. Il implémente des mécanismes qui permettent de parser le XML et de manipuler ses données. Deux modules composent Mexpat dans les librairies Mercury : `mexpat.m` et `mexpat.xml.m`. En les important dans un programme, on peut exploiter les fonctionnalités reprises dans leur interface.

Mexpat est en fait une adaptation de *expat* pour Mercury. Expat a été conçu en code *C* par James Clark[9] et est, tout comme Mexpat, une librairie pour le parsing de documents XML [12]. Cette section présente de manière simplifiée la syntaxe que Mexpat utilise pour représenter un document XML.

SYNTAXE :

▷ Un élément XML est représenté de la manière suivante :
`elem(string, list(attribute), list(content))`
Le string représente le nom de l'élément, le deuxième argument est la liste des attributs de cet élément et le troisième représente le contenu confiné entre les balises de l'élément.

▷ Le type "attribute" a la forme :
`attr(string, string)`
Le premier string représente le nom de l'attribut et le second sa valeur.

▷ Enfin "content" :
`celem(element)` ou `ctext(string)`
Le contenu d'un élément peut ainsi être composé d'autres éléments XML et de texte.

[9] James Clark a également travaillé dans l'équipe qui a produit la spécifiaction XML du W3C

EXEMPLE :

Le code XML :

```
<balise>
  Du texte 1
    <central attrib="hello" />
  Du texte 2
</balise>
```

est représenté en Mexpat par :

```
elem("balise", [], [
                     ctext("Du texte 1"),
                     celem("central", [attr("attrib", "hello")], []),
                     ctext("Du texte 2")
                    ]
)
```

Une fois qu'un document XML a été parsé par Mexpat, son contenu est incorporé dans la syntaxe exposée ci-dessus[10]. Il est alors possible de recourir aux mécanismes de manipulation des données. Par exemple, on peut demander à effectuer des recherches du style : récupérer la liste des éléments nommés "livre" qui ont pour grand-père un élément nommé "oeuvre" possédant un attribut "langue" de valeur "Français".

Mexpat permet ainsi de rechercher, récupérer et manipuler les données d'un document XML dans un programme Mercury.

[10]Cela ne ressort peut-être pas très bien, mais un document XML peut être vu comme un arbre avec un élément racine et où chaque élément peut avoir des enfants (soit textuels, soit d'autres éléments).

4.3 Resource Description Framework (RDF)

RDF est un langage du W3C particulièrement approprié pour représenter des informations sur le Web. Il constitue d'ailleurs un élément essentiel du Web sémantique.

"*Le Web sémantique désigne un ensemble de technologies visant à rendre le contenu des ressources du World Wide Web accessible et utilisable par les programmes et agents logiciels, grâce à un système de métadonnées*[11] *formelles, utilisant notamment la famille de langages développés par le W3C*" [3].

Ainsi l'idée de RDF est de procurer un langage formel qui permette à un software de pouvoir recueillir toutes sortes d'informations sur un site, de les comprendre et de les traiter. On le connait principalement dans le contexte du Web pour réprésenter des métadonnées, mais RDF permet (plus généralement) de modéliser des concepts et les relations qui les lient entre eux.

4.3.1 Les triplets

Un document RDF décrit des ressources[12] à l'aide de propriétés. La valeur d'une propriété peut être un string, un nombre, un format de date,... (c'est à dire une valeur terminale qui n'est pas une ressource) ou une autre ressource. Puisqu'une ressource peut en pointer une autre par l'intermédiaire d'une propriété, RDF permet de tracer des graphes de relations entre les ressources.

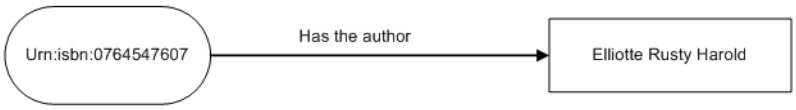


FIG. 4.1 – Triplet RDF

La figure 4.1 (issue du livre *XML Bible* [17]) montre une ressource (un numéro ISBN) dont l'auteur (propriété de la ressource) est Eliotte Rusty Harold.

[11] Les métadonnées sont des données sur les données [17]. Par exemple si une page Web constitue les données, le nom de l'auteur, la date de création de la page, l'information sur le contenu... constituent des métadonnées.

[12] Wikipedia traduit la définition d'une ressource (depuis la **RFC 2396**, aout 1998) comme étant '...toute chose qui possède une identité. Des exemples familiers incluent un document électronique, une image, un service (par exemple "le bulletin météo d'aujourd'hui pour Los Angeles"), ou un ensemble d'autres ressources. Certaines ressources ne peuvent pas être "ramenées par le réseau" (network retrievable), par exemple les êtres humains, les entreprises, les livres d'une bibliothèque peuvent être aussi considérés comme des ressources'.

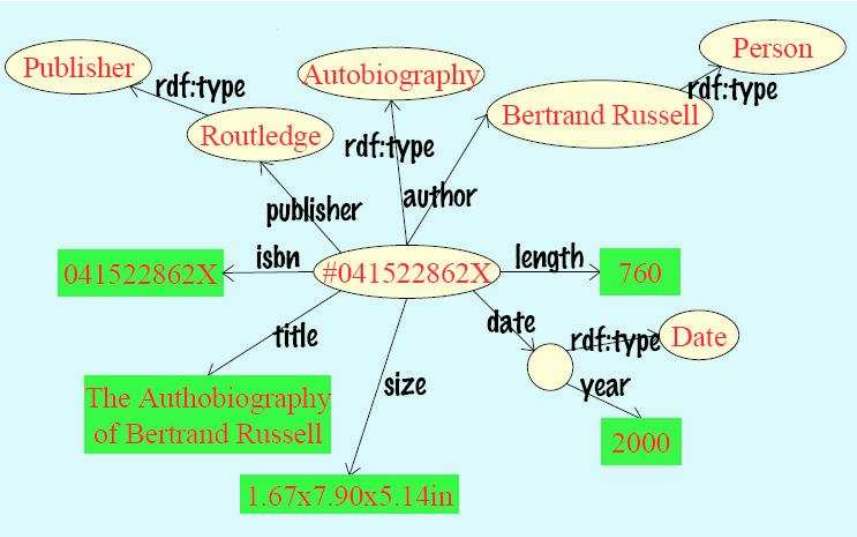


FIG. 4.2 – Schéma RDF

La figure 4.2 (provenant du cours [20]), légèrement plus complète, illustre une ressource qui s'avère aussi être un livre. Elle possède une multitude de propriétés qui pointent soit vers une valeur terminale (fond vert), soit vers une autre ressource (fond blanc).

Les composantes d'un triplet sont respectivement appelées *Sujet*, *Prédicat* et *Objet*. Ainsi, un document RDF est un ensemble de triplets de la forme

{sujet, predicat, objet}

où [3] :

- le sujet représente la ressource à décrire
- le prédicat représente le type de propriété appliqué à la ressource
- l'objet représente la valeur de la propriété pour cette ressource (c'est-à-dire une donnée ou une autre ressource)

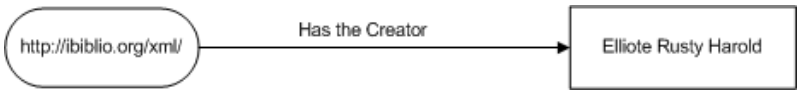
4.3.2 La syntaxe XML

La syntaxe RDF/XML a été développée par le W3C. Ce format standard a pour intérêt de rendre un ordinateur capable de parser un document RDF [17]. La plupart du temps, *Resource Data Framework* est associé à sa syntaxe XML, mais d'autres représentations existent cependant. "RDF est simplement une structure de données constituée de noeuds et organisée en graphe" [3]. De ce fait, le concept RDF n'est pas intrinsèquement lié à une syntaxe particulière[13].

Les éléments RDF (dans la représentation XML) appartiennent tous à l'espace de nom "`http://www.w3.org/1999/02/22-rdf-syntax-ns#`". L'élément racine d'un tel document doit être nommé "rdf :RDF"[14] (où *rdf* est associé à l'espace de nom précédemment cité). Une ressource est encodée grâce à l'élément "Description" auquel on greffe à l'attribut "about" pour lui associer l'URI qui représente le sujet.

EXEMPLE :

La phrase "Elliote Rusty Harold a créé le site web `http://ibiblio.org/xml/`" peut être représentée par le schéma :



La traduction du schéma en RDF/XML donne alors [17] :

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://ibiblio.org/xml/">
    <dc:creator>Elliote Rusty Harold</dc:creator>
  </rdf:Description>
</rdf:RDF>
```

Les propriétés d'une ressource sont représentées comme des éléments enfant de sa balise. Si l'on se réfère à l'espace de nom auquel est associé "dc", *creator* désigne "la personne ou l'organisation qui a créé la plupart de la ressource" [17] (traduction non officielle).

[13] La conversion d'un graphe RDF en un document RDF/XML n'est possible que si les noms des propriétés du graphe ne rentrent pas en conflit avec la syntaxe

[14] Nous ne considérons pas ici la syntaxe RDF/XML abrégée

Ceci constitue un exemple de base de la syntaxe, la spécification de la représentation XML du langage est décrite en détail par le W3C [31]. On y trouve notamment (en plus de toutes les spécificités que nous n'avons pas considérées) les abréviations possibles dans le RDF/XML.

Chapitre 5

MSP

MSP (acronyme de *Mercury Server Page*) est une solution générique et extensible pour la génération dynamique de XML [9]. La technologie Mercury Server Page se compose d'un langage de template accompagné d'un compilateur qui convertit les templates en code Mercury. Ce chapitre décrit le langage, donne des exemples d'utilisation et explique le fonctionnement global de la technologie.

L'utilisation typique de MSP intervient dans la création d'interfaces graphiques sur base d'un langage XML où la génération de l'interface dépend d'un contexte dynamique. Imaginons par exemple une page XHTML qui affiche les noms de toutes les personnes prénommées Henri contenues dans un fichier RDF/XML [1] (lequel est régulièrement remis à jour). Vu que le contenu du fichier est changeant, il faut pouvoir tenir compte de ce dynamisme dans la génération de l'interface : on ne peut pas créer un fichier XML statique à l'avance. D'autre part, la requête RDF est une manipulation qui requiert des connaissances en programmation. Il en découle qu'un graphiste seul n'est pas en mesure de réaliser le travail. Grâce à MSP, l'insertion de quelques balises (appartenant au langage de template) va permettre de récupérer toutes les personnes prénommées *Henri* du fichier RDF/XML "PersonSet.xml".

Il reste alors au designer à insérer son langage entre les balises MSP [2] concernées pour que son travail soit appliqué à chaque résultat de la requête RDF :

[1] cf. section 4.3.2

[2] la syntaxe de Mercury Server Page sera découverte progressivement tout au long de ce chapitre, il n'est pas nécessaire de comprendre la signification et l'intérêt de chaque balise à ce point du mémoire. L'exemple ici n'est présenté que pour illustrer l'intérêt de la technologie

```
<msp:all variables="X">
   <msp:goal>
      <msp:triple datasource="PersonSet.xml"
           subject="${X}" predicate="FirstName" object="Henri" />
   </msp:goal>
   <msp:content>
```

Insérer ici le XHTML qui concerne les personnes nommées Henri

```
   <msp:content>
</msp:all>
```

A partir de ce moment, le designer ne doit plus se soucier de la programmation et du raisonnement logique qui se cachent derrière l'opération réalisée. La seule chose qu'il sait est que la variable `X` va représenter successivement chaque personne de "PersonSet.xml" prénommée Henri.

EXEMPLE :

```
<msp:all variables="X">
   <msp:goal>
      <msp:triple datasource="PersonSet.xml"
           subject="${X}" predicate="FirstName" object="Henri" />
   </msp:goal>
   <msp:content>
```

<HR align="center" size="8" width="50%"/>
<H2 align="center"> ${X} se prénomme Henri ! </H2>

```
   <msp:content>
</msp:all>
```

Si la source RDF contient deux triples dont l'objet du prédicat *FirstName* est *Henri*, par exemple :

$\{CharlesHenri, FirstName, Henri\}$ et $\{VandenputHenri, FirstName, Henri\}$

Il en résultera :

```
<HR align="center" size="8" width="50%"/>
<H2 align="center"> CharlesHenri se prénomme Henri ! </H2>

<HR align="center" size="8" width="50%"/>
<H2 align="center"> VandenputHenri se prénomme Henri ! </H2>
```

Les tags qui ne débutent pas par "msp:" ne sont pas connus du langage, dans ce cas ils seront recopiés tels quels dans le résultat final.

Mercury est omniprésent dans MSP : C'est le langage dans lequel a été écrit le compilateur, c'est aussi le langage cible de ce *mspcompiler* et l'on peut même retrouver certaines incursions de Mercury dans le langage de template.

Revenons quelques instants au problème posé dans l'introduction (cf. page 8). Une solution possible était d'élaborer un langage de template capable, d'une part, de générer un résultat pour n'importe quel code de syntaxe XML, et d'autre part qui puisse de manière simplifiée faire appel à une logique de programmation implémentée séparément. C'est ce qui a été fait avec MSP.

Le développeur fournit son appui en cas de problème mais de manière générale son intervention est limitée. Le designer doit pouvoir créer lui-même ses interfaces en utilisant son langage de norme XML au sein d'un template. De même, si la structure d'un template est à sa disposition, il devrait être capable d'en effectuer la maintenance de manière relativement indépendante. MSP est en fait le point d'accord entre le spécialiste du langage cible et le développeur. Le design, le dynamisme de l'interface, et la facilité de la maintenance sont directement affectés par ses effets.

Cette utilisation typique de MSP (dans le domaine du design consistant à rendre les tâches séparables pour réattribuer à chacun son travail) n'est cependant pas l'unique service que peut rendre le langage de template. Ce n'est qu'une vue limitée car ce dernier offre un large éventail de possibilités : pour autant que le langage soit XML, MSP ne fait pas de restrictions sur son choix. Au lieu de générer de l'interface, on pourrait par exemple récupérer des données et les stocker dans un fichier XML, ou on pourrait extraire certaines informations nécessaires pour formuler une requête à un serveur communiquant sur base d'un protocole XML,... Les utilisations possibles sont aussi nombreuses que variées.

De manière plus générale, MSP procure des documents XML[3] bien formés et génère des résultats purement XML. Cette technologie permet un mélange d'éléments statiques et dynamiques, elle est construite sur des bases logiques,

[3]Les documents dont il est question ici sont les documents donnés en input à *mspcompiler*, pas le résultat généré.

cache la complexité du raisonnement et offre une solution générique par le biais d'un langage déclaratif simple.

5.1 La compilation

La génération du résultat sur base de la source MSP se voyait offrir deux alternatives : la compilation ou l'interprétation.

Un compilateur traduit un code source vers un code cible (rédigé dans un langage plus proche de la machine). Il effectue une analyse lexicale, syntaxique, puis sémantique avant de générer le code final [27] (qui est généralement amené à être exécuté). Un interpréteur réalise un travail similaire si ce n'est qu'il ne génère pas de code, il interprète directement les données et produit un résultat. D'un point de vue d'efficacité le premier procédé est plus performant à l'exécution car l'étape de compilation débouche sur un code de plus bas niveau. En effet, l'interprétation implique lors de l'exécution la répétition des étapes d'analyse du code (travail que le compilateur a déja effectué avant le *runtime*) et la conversion du code en instructions compréhensibles pour le processeur (ou dans un autre langage cible).

L'efficacité n'est cependant pas la raison qui justifie le choix d'avoir écrit un compilateur pour le langage de template MSP. Etant donné que Mercury est à la base du projet et qu'il n'existe pas d'interpréteur pour ce langage, il aurait fallu en écrire un. Ce travail aurait été colossal : Mercury est un langage complexe et son compilateur (qui ne cesse d'évoluer) est loin de représenter un exercice trivial.

C'est ainsi que le projet s'est orienté vers un compilateur qui relègue une partie du travail[4] au MMC (Mebourne Mercury Compiler). L'idée était de pouvoir réutiliser la puissance d'un langage de haut niveau préexistant pour se décharger de l'implémentation des bases du compilateur (gestion des types, des librairies,...) où Mercury est déjà bien optimisé. Si cette approche représente un avantage dans la "sous-traitance" des tâches, elle engendre aussi inévitablement des conséquences. *Mspcompiler* doit convertir les sources MSP en Mercury et donc tenir compte de la rigueur du compilateur de ce langage. De plus, étant donné que MMC effectue une partie du travail, il peut déclencher ses propres erreurs de compilation (cf. section 5.7 page 79).

Comme l'exprime le schéma de la figure 5.1, le fichier MSP d'entrée est compilé par *mspcompiler* qui produit en sortie un fichier intermédiaire "*file.m*". Il s'agit d'un fichier d'extension "*.m*" contenant une traduction des instructions de la source MSP en Mercury. Il faut alors le passer au compilateur Mercury[5]

[4]Les sources MSP sont converties en Mercury (qui reste un langage de haut niveau), MMC prend le relais ensuite.

[5]MSP fonctionne sous la version 0.13.1 du compilateur Mercury

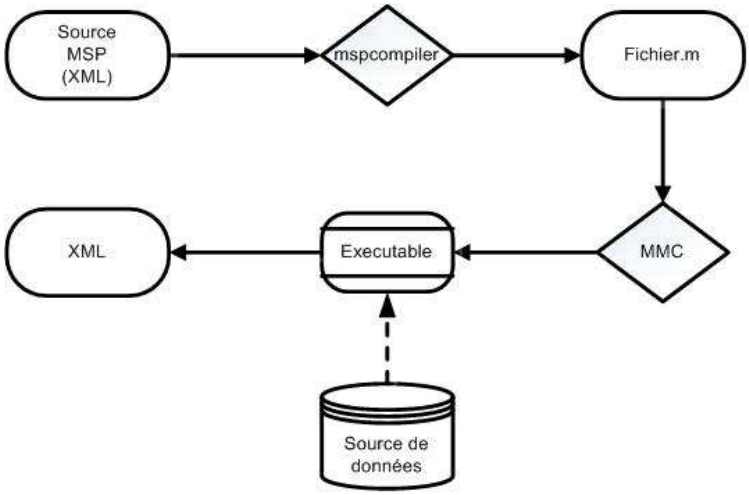


FIG. 5.1 – Les étapes de compilation

pour qu'il génère un exécutable. C'est seulement l'exécution de ce dernier qui produira le code XML[6].

Dans le cadre de ce mémoire, le terme "source MSP" ou "fichier MSP d'entrée" sera employé pour identifier le fichier passé au compilateur. Un fichier MSP d'entrée est un document dans lequel le code du langage cible (de syntaxe XML) est balisé par des tags. En effet, pour être strictement XML les fonctionnalités MSP sont renfermées dans des tags (ce qui permet également de délimiter leur portée).

Schématiquement, *mspcompiler* fait tout d'abord appel à un prédicat de mexpat (cf. section 4.2) qui va parser le document XML d'entrée, le transformer en arbre (A_1) et fournir une variable contenant la racine de ce dernier. Il applique alors sur la racine un prédicat qui va traiter récursivement chaque noeud de A1 pour créer un nouvel arbre (A_2).

La transformation de A_1 en A_2 est une passe du compilateur qui vise à préparer la conversion des instructions MSP en Mercury. A_1 représente la structure XML de la source MSP dans la syntaxe Mexpat tandis que A_2 est préformaté pour la transformation ultime. Non seulement la difficulté de transformation est réduite et l'information exprimée dans l'arbre est plus claire, mais il en découle également un gain de modularité : la transformation syntaxique est découplée de la conversion en code.

[6]potentiellement en consultant des données externes

Finalement A_2 est parcouru pour être converti en Mercury. Il en résultera un fichier d'extension *.m*. La section 5.5 présentera un exemple complet d'utilisation de MSP, allant du fichier d'entrée au résultat XML généré.

Les sections qui suivent vont décrire le langage de template. Ce dernier renferme des *fonctionnalités* divisées en trois catégories : **les instructions, les buts et les directives**. Quel que soit le groupe auquel elle appartient, une fonctionnalité MSP se discerne dans le fichier source par sa forme "`<msp:...>`". L'espace de nom qu'il faut associer à "`msp:`" est :

```
xmlns:msp="http://msp.missioncriticalit.com"
```

5.2 Les instructions MSP

Une instruction pourrait être définie comme une fonctionnalité qui offre un mécanisme ayant un effet direct dans la construction du XML sortant. Elle est interprétée par *mspcompiler* qui lui associe une action et génère une partie du résultat.

Dans cette section chacune des instructions existantes [7] sera introduite par une description accompagnée d'un ou plusieurs exemples explicatifs[8]. Les sept instructions qui vont être retrouvées dans les pages suivantes sont :

- `MSP : Forall`
- `MSP : If`
- `MSP : Switch`
- `MSP : Element`
- `MSP : All`
- `MSP : Template`
- `MSP : Attribute`

[7] A la date du 31 décembre 2006

[8] Les exemples donnés ont souvent XHTML pour langage cible mais n'importe quel autre langage XML aurait pu être employé

5.2.1 MSP : Forall

Le principe de l'instruction Forall est de reproduire son contenu (c'est-à-dire l'information contenue entre ses balises) pour chaque valeur que la variable prend (c'est-à-dire pour chaque élément de la liste). Un tag `MSP:Forall` contient deux attributs : `variable` et `in_list`. La liste est de type "list(T)"[9] et la variable a le type T. De cette manière la variable va pouvoir successivement adopter chacune des valeurs des éléments de la liste.

EXEMPLE :

```
<msp:forall variable="X" in_list="[1,2]">
   Un peu de contenu avec ${X} au milieu
</msp:forall>
```

RESULTAT :

```
   Un peu de contenu avec 1 au milieu
   Un peu de contenu avec 2 au milieu
```

L'apparition de variables est très courante dans les instructions MSP. Lors de la génération du code, l'occurrence de `${VAR}` sera remplacée par la valeur de la variable `VAR`. La forme sans accolade est d'application lorsqu'on la requiert explicitement (comme dans le cas du Forall) ou lorsqu'elle est intégrée dans du code Mercury. Le raisonnement logique étant opéré par le Mercury, les noms de variables doivent impérativement commencer par une majuscule. En réalité `${}` peut renfermer n'importe quelle expression Mercury.

[9] voir section 4.1.2

5.2.2 MSP : If

Le `msp:if` se divise en trois blocs : Condition, Then et Else. Si la condition est respectée, le contenu placé entre les balises du `msp:then` sera généré et dans le cas contraire, c'est le `msp:else` qui sera produit. Ce dernier bloc est facultatif, en cas d'absence du `msp:else` *mspcompiler* se comporte comme s'il avait rencontré un bloc vide.

SYNOPSIS :

```
<msp:if>
  <msp:condition>
     GOAL
  </msp:condition>
  <msp:then>
   Ce qui doit être généré en cas de succès de la condition
  </msp:then>
  <msp:else>
   Ce qui doit être généré sinon
  </msp:else>
</msp:if>
```

Dans le bloc `msp:condition` il faut respecter la syntaxe du GOAL (ou *but MSP*, cf. section 5.3 page 71).

EXEMPLE :

```
<msp:if>
     <msp:condition>
         <msp:and>
             <msp:mercury goal="X=1, member(X, [1,2,3])"/>
             <msp:not>
                 <msp:mercury goal="member(1, [1,2,3])"/>
             </msp:not>
             <msp:or>
                 <msp:mercury goal="member(1, [1,2,3])"/>
                 <msp:mercury goal="member(1, [2,3])"/>
             </msp:or>
         </msp:and>
     </msp:condition>
     <msp:then>
         <p>Alors la condition est vérifiée et X vaut ${X}</p>
     </msp:then>
     <msp:else>
         <p>Sinon nous générerons ceci</p>
     </msp:else>
</msp:if>
```

RESULTAT :

```
<p>Sinon nous générerons ceci</p>
```

5.2.3 MSP : Switch

Le `switch` permet de couvrir des cas multiples et de générer un résultat en fonction de celui qui est rencontré. Cette instruction est constituée de blocs `msp:case`, chacun contenant un `msp:condition` et un `msp:content`.

Un `msp:switch` sera converti dans son "Mercury switch" correspondant. Pour cette raison, l'ensemble des `msp:case` doit couvrir tous les cas possibles et les domaines couverts par chaque `msp:case` ne peuvent pas se recouper. Par conséquent, chaque cas possible doit rentrer dans les conditions d'exactement un `msp:case`.

EXEMPLE :

```
<html xmlns:msp="http://msp.missioncriticalit.com" >
     <p> With the list : [yes(train), no, yes(car), yes(plane)]</p>
     <msp:forall variable="Travel"
              in_list='[yes("train"), no, yes("car"),yes("plane")]'>
     <msp:switch>
       <msp:case>
           <msp:condition>
              <msp:mercury goal="Travel = yes(Means)"/>
           </msp:condition>
           <msp:content>
              <p>This one travels by ${Means}</p>
           </msp:content>
       </msp:case>
       <msp:case>
           <msp:condition>
              <msp:mercury goal="Travel = no"/>
           </msp:condition>
           <msp:content>
              <p>This one does not travel</p>
           </msp:content>
       </msp:case>
     </msp:switch>
     </msp:forall>
  </html>
```

RESULTAT :

```
<html xmlns:msp="http://msp.missioncriticalit.com">
<p> With the list : [yes(train), no, yes(car), yes(plane)] </p>
<p>This one travels by train</p>
<p>This one does not travel</p>
<p>This one travels by car</p>
<p>This one travels by plane</p>
</html>
```

5.2.4 MSP : Element

Il permet la création d'un tag dont le nom sera déterminé lors du *runtime* par la valeur d'une variable. De cette manière le nom de la balise sera attribué dynamiquement. L'intérêt de cette instruction est qu'elle est correcte sur le plan XML car quelque chose comme "<${VAR} >" serait refusé durant le parsing du document.

EXEMPLE :

```
<msp:forall variable="X" in_list='["MyElement"]'>
   <msp:element name="${X}">
      <p> Voici le contenu du nouvel élément </p>
   </msp:element>
</msp:forall>
```

RESULTAT :

```
<MyElement>
    <p> Voici le contenu du nouvel élément </p>
</MyElement>
```

5.2.5 MSP : All

Cas Général

Le "All" est constitué de deux blocs : un bloc de but (qui doit respecter la syntaxe de GOAL, voir section 5.3 page 71), et un bloc de contenu. Contrairement au `msp:if`, dans le cas du "All" le goal peut être non déterministe. Le contenu sera répété pour chaque solution venant du bloc de but. Des solutions multiples sont donc autorisées mais si le goal échoue , le bloc entier du `msp:all` sera ignoré.

EXEMPLE 1 :

```
<msp:all variables="X">
  <msp:goal>
     <msp:mercury goal="member(X, [1,2])"/>
  </msp:goal>
  <msp:content>
     X vaut ${X}
  </msp:content>
</msp:all>
```

RESULTAT 1 :

```
X vaut 1
X vaut 2
```

Tri Des Solutions

En ajoutant un attribut à l'intérieur de la balise `msp:all` il est possible de trier les solutions dans un ordre spécifique[10]. Les attributs `ascending-order="..."` et `descending-order="..."` permettent de ranger les résultats suivant une variable ou une fonction Mercury existante. Il est aussi possible de définir une fonction de tri à l'aide des attributs `ascending-compare="..."` et `descending-compare="..."`. Les deux exemples ci-après illustrent une utilisation possible de ces mécanismes.

[10]En effet, Mercury permet de faire des comparaisons avec tous les types, et donc de les trier

EXEMPLE 2 :

```
<msp:all variables="S" descending-order="length(S):int" >
   <msp:goal>
     <msp:mercury goal='member(S,
                 ["shortString",
                  "Very Very looooooooooooooooooong string",
                  "Here is a medium string"])'/>
   </msp:goal>
   <msp:content>
     ${S}
   </msp:content>
</msp:all>
```

RESULTAT 2 :

```
Very Very looooooooooooooooooong string
Here is a medium string
shortString
```

EXEMPLE 3 :

```
<msp:all variables="X, Y"
    ascending-compare='func({X1,Y1},{X2,Y2}) =
      ( Y1 < Y2 -> (>) ; Y1 = Y2 -> (=) ; (<) )' >
  <msp:goal>
     <msp:and>
       <msp:mercury goal="member(X, [1,2])"/>
       <msp:mercury goal="member(Y, [3,4])"/>
     </msp:and>
  </msp:goal>
  <msp:content>
     X is ${X} and Y is ${Y}
  </msp:content>
</msp:all>
```

RESULTAT 3 :

```
X is 1 and Y is 4
X is 2 and Y is 4
X is 1 and Y is 3
X is 2 and Y is 3
```

Lors de l'utilisation de *ascending-compare* ou de *descendig-compare* il est absolument nécessaire de créer la signature de la fonction en accord avec le nombre de variables déclarées. Cette dernière doit prendre comme arguments deux tuples Mercury contenant toutes les variables. Par exemple :

- `variables="X,Y"=>func({X1,Y1},{X2,Y2})=...`
- `variables="X,Y,Z"=>func({X1,Y1,Z1},{X2,Y2,Z2})=...`
- `variables="X"=>func({A},{B})=...`

Etant donné qu'une fonction personnalisée est écrite dans un string XML, les caractères XML doivent être escapés. Ex : "<" devient "<".

5.2.6 MSP : Template et MSP : apply-template

Cas Général

Les instructions `msp:template` et `msp:apply-template` permettent respectivement de définir une fonction Mercury et d'appeler une fonction. La définition renferme le résultat à produire et les informations requises quant aux paramètres à passer lors de l'appel, tandis que l'appel proprement dit contiendra juste les valeurs nécessaires à l'application de la définition. Ce dernier peut ainsi prendre très peu de place dans une source MSP pour réaliser un ensemble potentiellement volumineux d'opérations plus ou moins complexes (ce qui représente un avantage considérable lorsqu'il faut le répéter de nombreuses fois).

Les instructions `msp:template` et `msp:apply-template` permettent donc de définir un template et d'appliquer un template prédéfini. Il ne faut cependant pas les confondre avec la définition d'un "template" que l'on fournit à *mspcompiler*, `msp:template` et `msp:apply-template` sont des instructions MSP qui retournent un contenu XML. Elles tirent leur nom du fait qu'elles génèrent du XML préformaté... en ce sens elles jouent un rôle de template. Pour éviter de mélanger les définitions, le terme "sous-template" sera désormais employé en référence à ce nouveau concept.

Syntaxiquement `msp:template` est constituée d'un nom, d'une liste de paramètres et d'un corps.

SYNOPSIS :

```
<msp:template name="name_of_the_template">
   <msp:parameter name="Parameter1" type="Parameter_type"/>
   ...
   <msp:body>
    <p> An XML content </p>
   </msp:body>
</msp:template>
```

A chaque appel il faudra passer les paramètres (conformément à la définition) et le tout sera remplacé par le résultat de l'exécution de la fonction correspondante.

```
<msp:apply-template name="name_of_the_template">
   <msp:parameter> Value of Parameter1 </msp:parameter>
   ...
</msp:apply-template>
```

Comme pour les directives (cf. section 5.4 page 74), les "sous-templates" doivent être des fils directs du tag racine. Une fois cette règle respectée, on peut les appliquer n'importe où dans le code.

Lors de la définition d'un "sous-template", le nom donné à un paramètre sera le nom à mentionner dans le corps de la définition pour qu'il soit remplacé par sa valeur lors de l'exécution (comme on le ferait avec une variable). En pratique, la fonction résultante intégrera dans sa signature les noms des paramètres déclarés. L'occurence d'un de ces noms au sein du corps de la fonction fera donc référence à la variable correspondante qui lui aura été passée. C'est aussi la raison pour laquelle il faut associer un type (Mercury) à chaque paramètre défini.

Vu qu'une fonction en Mercury doit commencer par une minuscule, il en est de même pour le nom d'un `msp:template`. Par contre, les paramètres étant des variables Mercury, ils doivent commencer par une majuscule. Dans un `msp:apply-template` les paramètres peuvent adopter trois formes :

- Du code Mercury `${}`
- Un attribut (cf. section 5.2.7 page 70)
- Du contenu habituel (texte, XML, `${}`, ou mélange des trois)

Les exemples ci-dessous couvriront les différents cas. Un paramètre ayant pour valeur du `${code}` Mercury seul sera effectivement considéré comme du code Mercury (cf. exemple 1). Cependant, une chaîne de caractères est considérée comme du contenu XML[11]. Si l'on ajoute des caractères (ne fut-ce qu'un espace) avant ou après `${}`, cela aura pour effet d'associer la valeur du paramètre au type "xml.content" (le contenu XML habituel). Par conséquent, un simple espace peut changer l'évaluation du type du paramètre, et lui faire ainsi perdre son statut de code Mercury. C'est pour cette raison que la définition des types doit être effectuée avec précaution en fonction de l'utilisation qu'il va en être faite.

Une autre remarque doit être soulevée sur l'application d'un "sous-template", *mspcompiler* doit savoir à l'avance quel type va être retourné. En effet, un "sous-template" peut renvoyer soit du contenu soit des attributs (voir section 5.2.7

[11] De manière générale lorsqu'on rencontre une chaîne de caractères dans le document source, elle est réinjectée telle quelle dans le résultat final. Il en va de même lorsqu'on parse le contenu des balises "paramètre" (en ce qui concerne le typage), les chaînes de caractères et les tags sont considérés comme du contenu XML.

page 70) et le mélange des deux types dans une même instruction est interdit (cf. section 5.7.2 page 80). Etant donné que le "sous-template" peut être défini dans un fichier externe (cf. "Fichiers Externes" à la page 66) il n'est pas toujours possible de vérifier sa définition.

Si le type de retour est "attribut", il faut le spécifier comme suit :

```
<msp:apply-template name="mon_template" returns="attributes">
```

EXEMPLE 1 :

```
<html xmlns="http://www.w3.org/1999/xhtml"
           xmlns:msp="http://msp.missioncriticalit.com" >

  <msp:template name="my_dear_template">
     <msp:parameter name="Name" type="string"/>
     <msp:parameter name="X" type="int"/>
     <msp:parameter name="Y" type="int"/>
     <msp:body>
       <p>${Name} said that the sum of...</p>
       <p>${X} and ${Y} is <b>${X+Y}</b>.</p>
     </msp:body>
  </msp:template>

  <head>
     <title> XHTML Example </title>
  </head>

  <body bgcolor="#FFFFFF" link="#000000" text="red">
     <p>This is an XHTML example</p>
     <br/>
     <msp:forall variable="X" in_list="[1,2]">
       <p>I can count: ${X}</p>
       <msp:forall variable="Y" in_list="[3,4]">

           <msp:apply-template name="my_dear_template">
               <msp:parameter>${"You and I"}</msp:parameter>
               <msp:parameter>${X}</msp:parameter>
               <msp:parameter>${Y}</msp:parameter>
           </msp:apply-template>

       </msp:forall>
     </msp:forall>
   </body>

 </html>
```

RESULTAT 1 :

```
<html xmlns="http://www.w3.org/1999/xhtml"
           xmlns:msp="http://msp.missioncriticalit.com" >
  <head>
    <title> XHTML Example </title>
  </head>
  <body bgcolor="#FFFFFF" link="#000000" text="red">
    <p>This is an XHTML example</p>
    <br/>
       <p>I can count: 1</p>

       <p>You and I said that the sum of...</p>
       <p>1 and 3 is <b>4</b>.</p>

       <p>You and I said that the sum of...</p>
       <p>1 and 4 is <b>5</b>.</p>

       <p>I can count: 2</p>

       <p>You and I said that the sum of...</p>
       <p>2 and 3 is <b>5</b>.</p>

       <p>You and I said that the sum of...</p>
       <p>2 and 4 is <b>6</b>.</p>
  </body>
  </html>
```

EXEMPLE 2 :

```
<html xmlns="http://www.w3.org/1999/xhtml"
                xmlns:msp="http://msp.missioncriticalit.com" >
   <msp:template name="my_dear_template2">
     <msp:parameter name="Content" type="list(xml.content)"/>
     <msp:parameter name="SomeAttributes" type="list(xml.attribute)"/>
     <msp:parameter name="FancyText" type="list(xml.content)"/>
     <msp:body>
        <font>
           ${Content} said that:
           <msp:forall variable="Attr" in_list="SomeAttributes">
               <msp:attribute name="${Attr ^ attr_name}"
                              value="${Attr ^ attr_value}"/>
           </msp:forall>
           ${FancyText}
        </font>
     </msp:body>
   </msp:template>
 <head>
    <title> XHTML Example </title>
 </head>
 <body bgcolor="#FFFFFF" link="#000000">
    <p>This is an XHTML example</p> <br/>
    <msp:forall variable="X" in_list='["Here is a simple text",
                    "and another one which is slighly longer"]'>
        <p>
            ${X}
            <msp:forall variable="Color" in_list='["black","red"]'>
               <msp:apply-template name="my_dear_template2">
                  <msp:parameter>
                     <b>You and I</b>
                     <em>(well, us!)</em>
                  </msp:parameter>
                  <msp:parameter>
                     <msp:attribute name="color" value="${Color}" />
                  </msp:parameter>
                  <msp:parameter>
                     the color of this text is ${Color}
              </msp:parameter>
               </msp:apply-template>
            </msp:forall>
        </p>
     </msp:forall>
  </body>
  </html>
```

RESULTAT 2 :

```
<html xmlns="http://www.w3.org/1999/xhtml"
         xmlns:msp="http://msp.missioncriticalit.com" >
<head>
   <title> XHTML Example </title>
</head>
<body bgcolor="#FFFFFF" link="#000000">
   <p>This is an XHTML example</p> <br/>
   <p>
      here is a simple text
      <font color="black">
         <b>You and I</b>
         <em>(well, us!)</em> said that:
         the color of this text is black
      </font>
      <font color="red">
         <b>You and I</b>
         <em>(well, us!)</em> said that:
         the color of this text is red
     </font>
  </p>
  <p>
    and another one which is slighly longer
    <font color="black">
       <b>You and I</b>
       <em>(well, us!)</em> said that:
       the color of this text is black
    </font>
    <font color="red">
       <b>You and I</b>
       <em>(well, us!)</em> said that:
       the color of this text is red
    </font>
 </p>
</body>
</html>
```

Modules

A l'aide de `msp:template` et `msp:apply-template` il est possible de définir et d'appeler des "sous-templates". Il serait intéressant de pouvoir les créer dans des fichiers externes et de les importer à volonté, dans n'importe quel template. Cela est rendu possible avec `msp:library` et `msp:import-module` (cf. "directives" à la section 5.4 page 74).

Créer un tel sous-template externe revient à créer une librairie Mercury. Vu que les sous-templates doivent être impérativement définis sous la racine, il est obligatoire qu'il y en ait une. A cet effet, employer `msp:library` comme tag racine d'un sous-template informera *mspcompiler* qu'il peut se limiter à traiter les `msp:template` sous sa racine et ignorer le reste. Il est aussi intéressant de mentionner que compiler une source MSP avec l'option "-l" empêchera le compilateur de créer une fonction "main" dans le fichier Mercury (cf. section 5.6 page 78).

EXEMPLE 3 (définition) :

Créer le fichier "make_html_lists.xml" et y écrire le code suivant :

```
<msp:library>
  <msp:template name="make_unordered_list">
    <msp:parameter name="Points" type="list(string)"/>
    <msp:body>
       <UL>
         <msp:forall variable="NewPoint" in_list="Points">
            <LI>${NewPoint}</LI>
         </msp:forall>
       </UL>
    </msp:body>
  </msp:template>

  <msp:template name="make_ordered_list">
    <msp:parameter name="Points" type="list(string)"/>
    <msp:body>
       <OL>
         <msp:forall variable="NewPoint" in_list="Points">
             <LI>${NewPoint}</LI>
         </msp:forall>
       </OL>
    </msp:body>
  </msp:template>
</msp:library>
```

Exécuter ensuite "mspcompiler -library make_html_lists.xml" pour créer le fichier librairie make_html_lists.m

EXEMPLE 3 (source MSP contenant l'appel) :

Créer le fichier "test_import.xml" et y écrire le code suivant :

```
<html \url{xmlns="http://www.w3.org/1999/xhtml"
         xmlns:msp="http://msp.missioncriticalit.com"} >
  <msp:import-module name="make_html_lists" />
  <p> The six most important animal classes are :</p>
  <msp:apply-template name="make_ordered_list">
     <msp:parameter>
         ${["Mammals", "Birds", "Fish", "Reptiles",
         "Amphibians", "Anthropods"]}
     </msp:parameter>
  </msp:apply-template>
</html>
```

RESULTAT 3 :

Compiler le fichier d'entrée "mspcompiler -ignore-blanks test_import.xml", appliquer MMC sur "test_import.m" et exécuter "./test_import". Cela produira le résultat :

```
<html xmlns="http://www.w3.org/1999/xhtml"
         xmlns:msp="http://msp.missioncriticalit.com" >
<p> The six most important animal classes are :</p>
<OL>
   <LI>Mammals</LI>
   <LI>Birds</LI>
   <LI>Fish</LI>
   <LI>Reptiles</LI>
   <LI>Amphibians</LI>
   <LI>Anthropods</LI>
</OL>
</html>
```

Il pourrait arriver que plusieurs sous-templates aient les mêmes noms et paramètres dans différentes librairies (c'est-à-dire, en termes Mercury, plusieurs fonctions ayant la même signature dans différents modules). Pour remédier à ce problème il est possible de spécifier le module (la librairie, fichier.m) qui contient la fonction désirée, et ce par le biais de la syntaxe suivante :

```
<msp:apply-template name="myTemplate" module="myModule">
```

Par exemple, modifier l'exemple précédent comme suit impliquerait de spécifier le module :

EXEMPLE 4 (définition) :

Le fichier "make_html_lists.xml" est seulement constitué du sous-template "make_list" qui génère une liste HTML **énumérée**.

```
<msp:library>
   <msp:template name="make_list">
      <msp:parameter name="Points" type="list(string)"/>
      <msp:body>
        <OL>
           <msp:forall variable="NewPoint" in_list="Points">
              <LI>${NewPoint}</LI>
           </msp:forall>
        </OL>
      </msp:body>
   </msp:template>
</msp:library>
```

EXEMPLE 4 (source MSP avec définition et appel) :

Définir dans "test_import.xml" un autre template "make_list" qui génère une liste HTML **ordonnée**.

```
<html xmlns="http://www.w3.org/1999/xhtml"
         xmlns:msp="http://msp.missioncriticalit.com" >
   <msp:template name="make_list">
      <msp:parameter name="Points" type="list(string)"/>
      <msp:body>
        <UL>
          <msp:forall variable="NewPoint" in_list="Points">
             <LI>${NewPoint}</LI>
          </msp:forall>
        </UL>
      </msp:body>
   </msp:template>

   <msp:import-module name="make_html_lists" />

   <p> The six most important animal classes are :</p>
   <msp:apply-template name="make_list" module="make_html_lists">
      <msp:parameter>
           ${["Mammals", "Birds", "Fish", "Reptiles",
           "Amphibians", "Anthropods"]}
      </msp:parameter>
   </msp:apply-template>
</html>
```

Compiler et exécuter, le résultat sera le même que dans l'exemple 3. Si la liste énumérée était désirée, il suffirait de spécifier module="test_import" à la place de "make_html_list" dans `msp:apply-template`.

5.2.7 MSP : Attribute

Cette instruction permet d'ajouter dynamiquement des attributs à un tag. L'aspect dynamique réside dans le nombre d'attributs qui peuvent être rajoutés et dans les noms des attributs qui seront déterminés lors de l'exécution. Avant cette instruction, leurs valeurs pouvaient déjà être rendues dynamiques comme ceci :

```
<Tag  attribute1="${VAR1}" attribute2="${VAR2}"/>
```

Néanmoins, avec cette approche le nom des attributs restait statique (dans ce cas ci *attribute1* et *attribute2*). Avec `msp:attribute`, les noms peuvent également être générés dynamiquement.

Les éléments débutant par "msp :" appartiennent au langage MSP et représentent une information pour *mspcompiler*. Après son exécution, ces éléments ne feront pas partie du résultat généré. Si `msp:attribute` est le fils d'une autre instruction, l'élément parent sera dans ce cas laissé de côté de telle sorte que le nouvel attribut s'accroche à la première balise ancêtre qui n'est pas une instruction.

EXEMPLE SIMPLE :

```
<label>
  <msp:forall variable="{A, B}"
              in_list='[{"french", "objet"}, {"english", "object"}]'>

    <msp:attribute name="${A}" value="${B}" />

  </msp:forall>
</label>
```

RESULTAT :

```
   <label french="objet" english="object">
```

Ce résultat n'aurait pu être obtenu avec un simple `msp:forall` car le nom des attributs aurait forcément dû être statique.

5.3 Les buts MSP

Un goal (ou but) MSP correspond en fin de compte à un but Mercury. Sa syntaxe comprend 5 possibilités, dont 3 à définition récursive :

- `<msp:mercury goal="..."/>`
- `<msp:not>GOAL</msp:not>`
- `<msp:or>GOALS</msp:or>`
- `<msp:and>GOALS</msp:and>`
- `<msp:triple datasource="..." subject="..." predicate="..." object="..."/>`

La signification du "not", "or" et "and" est intuitive mais attention, il ne faut pas oublier que Mercury ne fonctionne pas comme un langage impératif. Si plus d'un but est satisfait à l'intérieur d'une même balise "or", l'évaluation du "or" devient multidéterministe.

La fonctionnalité `msp:mercury` permet d'évaluer un but Mercury en l'inscrivant entre les guillemets de l'attribut "goal". Ceci a déjà été illustré dans l'exemple 3 de l'instruction `msp:all` (page 58).

Le but `msp:triple` quant à lui mérite une attention particulière, le point suivant y est entièrement consacré.

Querying RDF

Un des objectifs de MSP consistait à être générique[12], par exemple en permettant le querying de triples RDF. De cette manière il est possible de récupérer l'information d'une source de données RDF et de l'exploiter pour produire le XML final.

Deux types de sources de données RDF peuvent être interrogées pour la requête de triples : des fichiers RDF/XML ou une base de données via ODBC.

- `<msp:tripledatasource="myRdfXmlFile.xml".../>`
- `<msp:tripledatasource="myOdbcDataBaseName".../>`

Avant de les interroger, les sources de données concernées doivent être définies avec `msp:datasource` (cf. section 5.4 page 74). Toute source définie peut alors être soumise à des requêtes de triples (textuellement, à l'aide de variables ou éventuellement en mélangeant les deux).

`Exemple : <msp:triple datasource="DataSourceNumber${Number}" .../>`

[12] Il est actuellement possible d'effectuer des requêtes RDF, mais MSP est générique dans le sens où, demain, il pourrait aussi être amené à devoir questionner des bases de données, ou des données XML, ou...

Les triples seront sélectionnés par les conditions fixées sur le sujet, le prédicat et l'objet de la requête.

EXEMPLE DE SOURCE DE DONNEES (Persons.xml) :

```
<?xml version="1.0"?>
   <rdf:RDF xmlns:rdf="http://www.w3.org/1999/XSL/Transform\#rdf">

   <person about="john">
     <personname>John Doe</personname>
     <sex>male</sex>
     <role>systemsadministration</role>
   </person>

   <person about="paul">
     <personname>Paul Smith</personname>
     <sex>male</sex>
     <role>legalsecretary</role>
     <married>mary</married>
   </person>

   <person about="mary">
     <personname>Mary Mary</personname>
     <sex>female</sex>
     <role>businessterritorymanager</role>
     <married>paul</married>
   </person>

   <person about="carla">
     <personname>Carla brownie</personname>
     <sex>female</sex>
     <role>bus driver</role>
     <married>john</married>
   </person>

   </rdf:RDF>
```

EXEMPLE DE FICHIER SOURCE :

```
<Root>
   <msp:datasource type="rdf_file" source="Persons.xml" />
   This is the list of the women from PersonsRdf.xml who are married :
   <msp:all variables="X, Y">
      <msp:goal>
        <msp:and>
           <msp:triple datasource="Persons.xml"
                  subject="${X}"
                  predicate="sex"
                  object="female"/>
           <msp:triple datasource="Persons.xml"
                  subject="${X}"
                  predicate="married"
                  object="${Y}"/>
        </msp:and>
      </msp:goal>
      <msp:content>
          <p>${X} is married to ${Y}</p>
      </msp:content>
   </msp:all>
</Root>
```

RESULTAT :

```
This is the list of the women from PersonsRdf.xml who are married :

             <p>carla is married to john</p>

             <p>mary is married to paul</p>
```

Il faut souligner qu'une requête de triples RDF peut être réalisée par un template externe, c'est-à-dire un template importé (défini dans un fichier externe). Cependant, la déclaration de la source de données doit être faite dans la source MSP car *mspcompiler* ne connaît que les sources de données déclarées comme **enfants directs de la balise du fichier d'entrée**.

5.4 Les directives

Les fonctionnalités qui ne sont ni une instruction ni un goal sont des directives. Les directives n'ont pas d'effet en soi, elles représentent une information pour *mspcompiler*. Actuellement cette catégorie ne contient que deux éléments : `msp:import-module` et `msp:datasource`.

5.4.1 Importer des modules

Des modules externes peuvent être importés dans le fichier MSP d'entrée. Ces derniers peuvent être des modules Mercury existants ou n'importe quel *fichier.m* (potentiellement une librairie générée par *mspcompiler*[13], cf. section 5.2.6 page 60). Afin d'importer un module, il suffit d'insérer un tag `msp:import-module` comme fils direct de la racine (comme fils direct, pas dans la descendance).

5.4.2 La déclaration de sources de données RDF

La directive `msp:datasource` permet de déclarer une source de données RDF (nécessaire avant les requêtes `<msp:triple...>`). Une fois de plus les déclarations des sources doivent être enfant du tag racine, sans quoi les requêtes la concernant seraient ignorées. Si la source est un fichier, son chemin d'accès doit être spécifié dans l'attribut "source" avec le type `"rdf_file"`. En ce qui concerne ODBC, le type doit être fixé à `"rdf_odbc"` et le champs "source" doit contenir le nom associé à la base de données en question. Dans ce dernier cas, il est obligatoire de spécifier un nom d'utilisateur et un mot de passe.

- `<msp:datasource type="rdf_file" source="myRdfXmlFile.xml"/>`
- `<msp:datasource type="rdf_odbc" source="myOdbcDataBaseName" userid="MyId"pwd="ThePassword"/>`[14]

[13] En effet, nous avons vu que l'instruction `msp:template` revenait à définir une fonction Mercury. Il est alors possible de créer un fichier qui ne contient que des définitions de fonction (accompagnées leur implémentation) et pas de prédicat *main*. Le fichier Mercury ainsi généré peut être importé comme n'importe quel autre module et faire office de librairie.

[14] Laisser le mot de passe en clair n'est certes pas sécurisé, il s'agit d'une solution provisoire.

5.5 Un exemple complet de MSP

Comme cela a été expliqué à la page 49, lorsque *mspcompiler* est exécuté avec un fichier source il commence par appeler Mexpat. Il en résulte un arbre[15] (A_1) qui est parcouru et transformé en un autre arbre (A_2) proche de la sémantique MSP. Cet arbre A_2 est alors lui-même converti en code Mercury, ce qui constitue le fichier de sortie de *mspcompiler*. Nous allons voir ici un exemple simple mais complet d'utilisation de MSP (c'est-à-dire la source XML, sa transformation Mercury, et son résultat XML).

Prenons le cas d'un `msp:forall` :

```
<Domaine>
  Un entier peut etre :
  <UL>
  <msp:forall variable="Entier" in_list='["Negatif", "Nul", "Positif"]' >
     <LI> ${Entier} </LI>
  </msp:forall>
  </UL>
</Domaine>
```

Le compilateur MSP génèrera sur base de cette entrée un fichier Mercury (cf. page suivante). Le *fichier.m* sera compilé par MMC et l'exécution du résultat de cette dernière compilation produira le XML (avec une liste XHTML) :

```
<Domaine>
  Un entier peut etre :
  <UL>
     <LI> Negatif </LI>
     <LI> Nul </LI>
     <LI> Positif </LI>
  </UL>
</Domaine>
```

Le fichier Mercury généré entre la source et le résultat comporte un grand nombre de lignes malgré la petite opération demandée. Cela s'explique par le fait que le début du fichier repose sur une structure principalement statique d'une cinquantaine de lignes. Nous ne justifierons pas ici tous les imports (ni les lignes de codes qui implémentent le prédicat main), le but est de montrer à quoi ressemble un fichier output de *mspcompiler* et d'expliquer comment le *forall* est converti en Mercury (dans la partie *implémentation*[16] du fichier).

[15] écrit dans la syntaxe Mexpat, cf. section 4.2
[16] cf. page 37

LE FICHIER MERCURY (généré par *mspcompiler*) :

```
%-----------------------------------------------------------------------
%***********************************************************************
%-----------------------------------------------------------------------

:- module 'forallTupleTry'.

:- interface.
:- import_module mexpat.xml.
:- import_module string.
:- import_module mspmodule.
:- import_module int.
:- import_module list.
:- import_module maybe.
:- import_module pair.
:- import_module io.
:- import_module map.
:- import_module rdfStore.
%-----------------------------------------------------------------------
%------------------------- TEMPLATE DECLARATIONS -----------------------
%-----------------------------------------------------------------------
:- pred main(io::di, io::uo) is cc_multi.

%-----------------------------------------------------------------------
%------------------ Declaration of the output function -----------------
%-----------------------------------------------------------------------
:- func output(map(string, list(rdfStore.triple))) = list(xml.content).

:- pragma source_file("forallTupleTry.xml").

%**************************** IMPLEMENTATION ***************************
:- implementation.

:- import_module mexpat.
:- import_module rdfmem.
:- import_module solutions.
:- import_module log4m.
%------------------------- TEMPLATE DEFINITIONS ------------------------

main(!IO) :-
    activate_logs(!IO),
    RdfFilesLst = [],
    list.map_foldl(build_triple_list(rdf_file), RdfFilesLst,
                   TriplesCorrList, !IO),
```

```
    RdfRefMap = map.from_corresponding_lists(RdfFilesLst, TriplesCorrList),
    write_element(nest_into_result_element_if_needed(
                 'forallTupleTry'.output(RdfRefMap)),!IO).

%---------------------- END OF TEMPLATE DEFINITIONS ---------------------
%------------------------------------------------------------------------

output(RdfRefMap) =
[celem(elem("Domaine", [],

     [ctext("Un entier peut etre :"),
     celem(elem("UL", [], condense(


        list.map( func(Entier) = [celem(elem("LI", [], to_xml(Entier)))],
                                  ["Negatif", "Nul", "Positif"] )

     )))

     ]

))] :- RdfRefMap = _.
```

La partie qui nous intéresse se trouve sous la démarcation "END OF TEMPLATE DEFINITIONS". On y trouve une fonction *output* qui renvoie le résultat XML. Pour rendre le code lisible et compréhensible, les informations sur les numéros de ligne (gestion des erreurs) ont été omises et la mise en page a été réorganisée.

Le `msp:forall` est implémenté à l'aide de la fonction Mercury *list.map*. Cette dernière prend en premier argument une fonction et en second lieu une liste. La fonction passée est appliquée à chaque élément de la liste et *list.map* renvoie une nouvelle liste (contenant les résultats successifs de l'opération). Sans rentrer dans les détails on constate que, dans notre cas, on passe à *list.map* une fonction qui applique un élément Mexpat (le `<LI>` de la source) à la liste `["Negatif","Nul", "Positif"]`. L'élément "LI" contient une variable (nommée "Entier") qui va successivement prendre la valeur de chacun des éléments de la liste d'entrée pour former chacun des résultats qui seront placés dans la liste de sortie (cf. le résultat de la page 75).

5.6 Les options du compilateur

Quatre options peuvent intervenir dans l'exécution de *mspcompiler* :

- -h (ou –help) affiche l'information sur les options disponibles.
- -o (ou –output) place l'output dans <file>.
- -l (ou –library) avertit *mspcompiler* que le sous-template qu'il doit compiler n'est qu'une librairie et qu'il ne doit pas générer de "main/2". Si cette option n'est pas utilisée et que le fichier est ensuite importé, le compilateur Mercury sera confronté à l'existence de deux prédicats "main/2" et une erreur de compilation sera lancée. L'option "-l" va être utilisée sur un sous-template constitué uniquement de définitions. Cependant il pourrait aussi être appliqué sur un sous-template classique... et aucune fonction "main" ne serait générée dans "fichier.m". L'utilisation de ce module serait alors limitée à une librairie et il ne pourrait pas être exécuté seul.
- -b (ou –ignore-blanks). Avec cette option, le compilateur MSP ignorera tous les "blancs pleins", c'est-à-dire les blancs qui n'ont aucun contenu (pas de caractères, pas de code Mercury...), de cette manière *mspcompiler* colle les unes contre les autres toutes les balises qui ne sont séparées que par des blancs (espaces, tabulations, retours à la ligne). Les conséquences sont importantes car sans cette option le moindre blanc entourant un tag `msp:attribute` est considéré comme du contenu... or mélanger des attributs et du contenu dans un sous-template est interdit (cf. section 5.7.2 page 80).

Cette option est particulièrement utile lorsque l'on recourt à `msp:attribute` en valeur de paramètre.

5.7 La gestion des erreurs

Les étapes à passer pour transformer une source MSP en XML (son résultat final) commencent par la compilation avec *mspcompiler* qui produit un fichier Mercury. Ce fichier (intermédiaire) doit ensuite être compilé avec MMC dont il résultera un exécutable. Enfin, l'exécution de ce dernier produira le résultat XML. Ces différentes étapes traversent 3 phases où des erreurs peuvent être soulevées :

- L'appel à Mexpat
- L'exécution de *mspcompiler*
- L'exécution du compilateur Mercury

Etant un parseur XML, Mexpat produira des erreurs si le document d'entrée n'est pas bien formé. Ensuite Le compilateur MSP s'assurera que la syntaxe du langage de template soit respectée et vérifiera qu'il n'y ait par d'instructions renvoyant à la fois du contenu et des attributs dynamiques (cf. 5.7.2). Enfin, MMC inspectera la sémantique Mercury (modes, types, déterminisme, instanciation des variables,...).

5.7.1 Implication du compilateur Mercury

L'intervention de deux compilateurs implique que les erreurs déclenchées peuvent provenir de sources différentes. Malgré ce problème, elles sont facilement discernables étant donné leur survenance à des moments différents dans les étapes de compilation. Une difficulté émerge tout de même de cette situation. Elle réside dans le report des erreurs vers une ligne du fichier MSP d'entrée lorsque ces dernières surviennent à la compilation du fichier Mercury. Dans ce cas, il faut en effet utiliser un mécanisme permettant de reparcourir le chemin en sens inverse, remontant les étapes pour référencer la bonne ligne dans la source MSP.

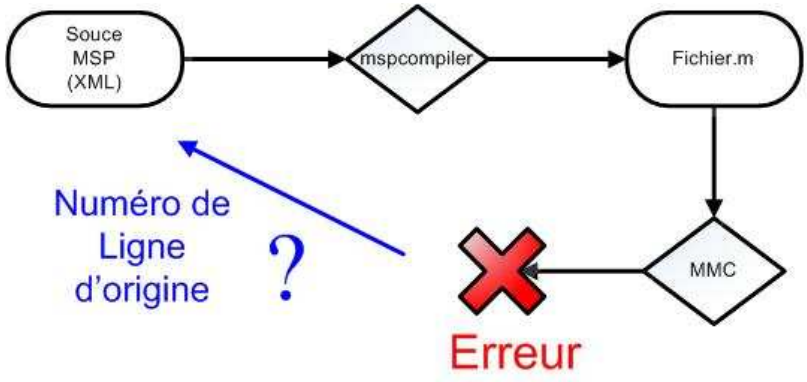


FIG. 5.2 – Le problème du report des erreurs

Il faut donc trouver un moyen de donner à MMC des références vers le fichier d'origine. Le problème est résolu à l'aide d'un mécanisme Mercury existant :

```
:- pragma source_file("multif.xml").
```

Après avoir posé cette ligne dans un fichier Mercury, il suffit d'insérer "#NuméroLigne" dans ce même fichier pour que MMC considère qu'il se trouve à la ligne "NuméroLigne" de la source qui a été mentionnée dans `source_file`. Le compilateur MSP génère ainsi dans le *fichier.m* une multitude de références aux numéros de lignes du fichier source.

Du point de vue de l'utilisateur, les erreurs mentionnent toujours un nom de fichier (et si possible un numéro de ligne). Avec le compilateur MSP il n'y a pas de grande difficulté à identifier une erreur. Dans le cas de MMC, la référence vers la source MSP (XML) devrait permettre à l'utilisateur de comprendre son erreur dans la plupart des cas. Si ce dernier est chevronné, il peut également se pencher sur les lignes d'erreurs précédées de "*fichier.m*" qui pourront lui fournir des indications supplémentaires (et éventuellement le pousser à plonger dans le code Mercury pour rechercher la cause de son problème).

5.7.2 Implication de mspcompiler

Il subsiste un inconvénient à tout cela : lorsque MMC génère une erreur, c'est une erreur de compilation Mercury. La signification et la cause de cette erreur ne sont pas toujours claires pour une personne qui n'est pas familière à Mercury. Dans un souci de clarté, *mspcompiler* détecte les erreurs de syntaxes MSP (comme l'oubli d'un attribut nécessaire à une instruction) et formule une erreur plus compréhensible pour un non-programmeur.

Un partie importante du problème étant laissée au MMC, *mspcompiler* soulève principalement des erreurs d'ordre syntaxique, mais il en est une qui ressort de l'ordinaire :

La plupart des instructions se traduisent par une génération de contenu XML. Cependant, étant donné que `msp:attribute` permet d'ajouter un attribut à un tag, deux types d'information différents doivent être gérés : du contenu XML (ce qui est retourné par la plupart des instructions) ou des attributs dynamiques (qui doivent alors être rajoutés à l'élément parent). Mais un attribut ne représente pas de contenu strictement parlant, nous avons dès lors deux types d'information différents. Il a été décidé qu'à l'intérieur d'un "msp:forall", "msp:if", "msp:all", "msp:switch" ou "msp:template", mélanger ces deux types est interdit. En effet certains tests[17] incluant des fonctions retournant un mix de contenu et d'attributs ont débouché sur des situations insensées. Les instructions MSP retournent pour cette raison soit du contenu, soit des attributs.

[17] Ces tests ont été menés par d'autres membres de l'équipe de projet MSP

5.8 MSP face aux autres solutions

Un des grands objectifs visés par MSP était d'être générique et c'est probablement ce qui le distingue le plus des technologies alternatives en matière de génération dynamique de XML. Bien que l'on puisse lui assigner un cas d'utilisation typique (à savoir la génération d'interface graphique), les usages que l'on peut en faire sont nombreux. De plus, étant extensible, la possibilité de lui greffer d'autres fonctionnalités dans le futur est avérée[18].

Contrairement aux technologies de la famille des *server pages*, MSP n'a pas été pensé pour travailler exclusivement avec un serveur Web (ce qui peut constituer une critique sur le choix de son nom). Cependant, il s'apparente clairement plus aux solutions présentées à la section 3.1 (dans le contexte du Web dynamique) qu'à XSLT. Rappelons que XSLT réalise de la transformation de documents XML, ce qui l'écarte conceptuellement de notre langage de template.

A la base, MSP se distingue aussi par l'approche déclarative qu'il adopte (plus intuitive pour un utilisateur lambda). Il faut tout de même nuancer cette affirmation car HSP n'est pas un langage impératif et JSP prend des allures déclaratives lorsque le code Java est totalement remplacé par des balises personnalisées.

La conception de *Mercury Server Page* s'est quelque peu inspirée des technologies du domaine pour s'agrémenter de fonctionnalités intéressantes dans le contexte de la génération dynamique de XML[19], mais toute la structure repose sur le langage Mercury qui représente la source d'inspiration maîtresse[20].

Parmi les solutions que nous avons parcourues au chapitre 3, celle qui se rapproche le plus de MSP est certainement Java Serveur Page (lorsqu'il est combiné à l'utilisation des balises personnalisées). En effet l'idée originelle est d'extraire au maximum la programmation (et la complexité de la logique) du fichier de base dont l'essence réelle est le XML. Sur ce point les autres technologies peuvent être la cause d'incursions monumentales de code étranger dans la source, ce qui nuit incontestablement à sa clarté.

[18] MSP a été imaginé pour être extensible et son développement s'est déroulé incrémentalement (par intégration successive de nouvelles fonctionnalités)

[19] par exemple `xsl:element` de XSLT qui permet de créer un tag dont le nom n'est pas statique

[20] Mercury est surtout la composante incontournable de MSP car l'entièreté de la logique de programmation y est subordonnée

Chapitre 6

Conclusion

6.1 Compte-rendu

XML est un incontournable de l'Internet, largement utilisé pour représenter l'information et la diffuser sur les réseaux. Si le Web semble être sa place de prédilection, ce n'est pas pour autant le seul domaine où on peut le rencontrer. Le besoin de génération dynamique de XML s'étend également à des horizons plus larges, preuve en est avec MSP qui fut conçu dans une optique complètement détachée du net.

Le langage de template et son compilateur répondent à la demande de *Mission Critical* qui désirait un langage particulièrement adapté à leur business. L'exemple introductif se voit offrir une solution puisque la génération d'interface graphique[1] avec dynamisme est simplifée. De surcroît, les objectifs fixés sur le projet ont été atteints : MSP est déclaratif, dynamique, générique, il permet des incursions de Mercury (qui offrent ainsi la possibilité de faire appel à des programmes externes) et la présence du code nécessaire à son fonctionnement n'est pas prédominante dans le document. Une personne n'ayant pas de connaissances en programmation pourrait éprouver des difficultés à utiliser certaines fonctionnalités du langage, mais les avantages précités lui permettront tout de même de travailler dans le document et de remplir sa tâche sans devoir comprendre la logique sous-jacente.

Les multiples usages qui peuvent être faits de MSP sont difficilement délimitables. Tant que le langage inséré dans le template se conforme à la norme XML, aucune restriction n'est imposée sur sa nature et sur l'objet de la génération finale. Parmi les fonctionnalités offertes, il est intéressant de rappeler qu'il est possible d'interroger une base de données RDF et d'insérer le résultat dans le XML généré. Cela illustre clairement que MSP n'est pas cantonné aux limites de son code, il peut interagir avec d'autres programmes et des données externes.

[1] sur base d'un langage XML

L'entreprise de développement logiciel travaillant avec Mercury, il était impensable de ne pas employer ce langage de programmation pour construire le compilateur. De plus, la sémantique de MSP se trouve relativement proche de Mercury, l'implémentation des instructions en est ainsi facilitée.

Mercury Server Page est le fruit d'un travail d'équipe qui a duré quatre mois. La collaboration de programmeurs expérimentés a permis d'améliorer la qualité du code tant en terme d'efficacité que de clarté et de concision. L'organisation s'est basée sur un système de "review" où toute production (ou modification) de code était vérifiée par un autre programmeur avant de pouvoir être "commitée" sur le serveur SVN où repose le projet. Cette méthode implique une bonne communication et un travail en équipe bien organisé qui ne peut qu'apporter des bénéfices à la qualité du développement.

Il faut aussi noter que le projet n'a pas été mené à titre d'expérience ou de recherche, le langage de template va être utilisé à des fins réelles par *Mission Critical*. Les fonctionnalités existantes continuent à être affinées, et d'autres idées devraient être implémentées dans le futur.

6.2 Travail futur

Le caractère extensible du langage de template ouvre de larges perspectives d'avenir pour le développement de fonctionnalités nouvelles. MSP pourra de cette manière évoluer en fonction des besoins rencontrés.

L'implémentation future la plus importante dans les intentions de développement concerne l'intégration de fonctionnalités OWL[2]. Ce langage basé sur la syntaxe RDF/XML [21] permet de définir des ontologies[3]. Dans notre cas, OWL serait utilisé comme un modèle d'instances RDF. Il serait alors intéressant de pouvoir interroger ce modèle pour l'exploiter (comme il est déja possible de le faire avec RDF) ou pour connaitre les propriétés des instances avec lesquelles on travaille.

Quelques autres idées visant à améliorer le langage ont été relevées. Parmi elles se trouve l'élaboration d'une instruction `"msp:any"` qui renvoie une seule solution (lorsqu'il existe au moins une solution) pour un but non déterministe, ou encore la mise en oeuvre d'une gestion des espaces de noms de manière à pouvoir choisir le préfixe des instructions au lieu de devoir obligatoirement recourir à `"msp:"`.

[2] Web Ontology language

[3] "Une ontologie est un ensemble structuré de concepts [...]. L'objectif premier d'une ontologie est de modéliser un ensemble de connaissances dans un domaine donné" [3]

En ce qui concerne son utilisation concrète, Mercury Server Page est amené à être intégré dans des logiciels plus complexes où il ne représentera qu'une composante de programmes beaucoup plus larges. Actuellement, il est déja testé dans des projets en cours de développement chez *Mission Critical*. Si son avenir n'est pas encore certain, il y a tout de même de fortes chances qu'un usage réel en soit fait dans le futur. MSP n'a probablement pas fini d'évoluer.

Acronymes

ASP : Active Server Page
CGI : Common Gateway Interface
DHTML : Dynamic HyperText Mark-up Language
DOM : Document Object Model
HSP : Haskell Server Page
HTML : HyperText Mark-up Language
JSP : Java Server Page
JSTL : Java server page Standard Tag Library
MMC : Melbourne Mercury Compiler
MSP : Mercury Server Page
OWL : Web Ontology Language
PHP : Personal Home Page
RDF : Resource Description Framework
SAX : Simple API for XML
SVN : SubVersioN
TLD : Tag Library Descriptor
URI : Uniform Resource Indentifier
URL : Uniform Resource Locator
URN : Uniform Resource Name
XHTML : eXtensible HyperText Mark-up Language
XML : eXtensible Mark-up Language
XSL : eXtensible Style Language
XSLT : eXtensible Style Language Transformation

Index

Bibliographie

[1] Comment Ca Marche l'informatique (CCM)
`http://www.commentcamarche.net/`.

[2] Cours de XML - Utilisation du DOM et XSLT dynamique
`http://gilles.chagnon.free.fr/cours/xml/domxslt.html`.

[3] Wikipedia, `http://www.wikipedia.org/`.

[4] *Site officiel du langage Mercury.*
`http://www.cs.mu.oz.au/research/mercury/index.html/`, 2007.

[5] M. Baron. *Cours - Java pour le développement d'applications Web : J2EE (JSP).* `http://mbaron.developpez.com/javaee/jsp/`, 2006.

[6] M. Baron. *Cours - Java pour le développement d'applications Web : J2EE (Taglib).* `http://mbaron.developpez.com/javaee/taglib/`, 2006.

[7] Ralph Becket. *Mercury Tutorial.* `http://www.cs.mu.oz.au/mercury/`, 2005.

[8] Niklas Broberg. *Thesis - Haskell Server Pages.* Chalmers University of Technology, 2005.

[9] Bastien De Buyser. *MSP Userguide.* Mission Critical, 2006.

[10] C. Ullman, D. Buser, J Duckett, B Francis, J. Kauffman, J.T. Llibre, D Sussman. *Initiation à ASP 3.0.* Eyrolles, Paris, 2000. Traduit de l'anglais par Florence Thierry, Geneviève Vassaux et Ingrid Pigueron.

[11] Isaac COHEN. *CGI/Perl et Javascript.* Eyrolles, Paris, 1996.

[12] Clark Cooper. *Using Expat.*
`http://www.xml.com/pub/a/1999/09/expat/index.html/`, 1999.

[13] F. Degrave. *Inférence sur RDF et génération de XML en Mercury.* FUNDP, 2006.

[14] Erik Meijer, Danny van Velzen. *Haskell Server Pages, Functional Programing and the Battle for the Middle Tier.* Electronic Notes in Theoretical Computer Science 41 No. 1 (2001).

[15] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, Chris Speirs, Tyson Dowd, Ralph Becket, Mark Brown. *The Mercury Language Reference Manual (Version 0.13.1).* The University Of Melbourne, 2006.

[16] P. Genevès. *Traitement des données et documents XML*. INRIA, Grenoble, France, 2006. (transparents de cours).

[17] Elliotte RUSTY HAROLD. *XML Bible*. Hungry minds, Inc., United States, 2nd edition, 2001. p.220.

[18] Elliotte RUSTY HAROLD and W. Scott MEANS. *XML en concentré*. O'RELLY, Paris, 3e edition, 2005. traduction de Philippe Ensarguet et Frédéric Laurent.

[19] James Weaver, Kevin Mukhar, Jim Crume. *J2EE 1.4*. Eyrolles, Paris, 2004. Adapté de l'anglais par Elvira Horvath.

[20] Jérôme Euzenat, Jean-François Baget. *Cours - Websémantique et représentation de connaissance*. `http://www.inrialpes.fr/exmo/teaching/swc/`, 2006.

[21] Francis Lapique. *Le Langage d'Ontologie Web OWL. FI 8*, pages 3–8, 24 octrobre 2006.

[22] J. Lemordant. *Cours XML*. INRIA, Grenoble, France.

[23] Benoît Marchal. *XML by example*. Que, Indianapolis, United States of America, 2nd edition, 2001.

[24] Jean-Luc Massat. *Cours - La technologie JSP (Java Server Page)*. `http://www.dil.univ-mrs.fr/~massat/ens/java/jsp1.html/`, Université de Luminy, 2007.

[25] M. Noirhomme-Fraiture. *cours INFO 2105 - Interactions Homme/Machine*. FUNDP, 2004.

[26] Christian Rémillard. *XSLT : Introduction et concepts. Architecture du dialecte*. `http://grds.ebsi.umontreal.ca/remillc/inu1011/2004/diapo-10-00.htm`, 2004.

[27] P.-Y. Schobbens. *cours INFO 2108 - Théorie des langages : syntaxe et sémantique*. FUNDP, 2004.

[28] Wim Vanhoof. *cours INFO 2109 - Programmation Fonctionnelle et Logique*. FUNDP, 2004.

[29] W3C. *XSL Transformations (XSLT) Version 1.0*. `http://www.w3.org/TR/1999/REC-xslt-19991116/`, 1999.

[30] W3C. *Extensible Markup Language (XML) 1.0 (Second Edition)*. `http://www.w3.org/TR/2000/REC-xml-20001006/`, 2000.

[31] W3C. *RDF/XML Syntax Specification (Revised)*. `http://www.w3.org/TR/rdf-syntax-grammar/#section-Infoset-Grammar/`, 2004.

[32] W3C. *Namespaces in XML 1.0 (Second Edition)*. `http://www.w3.org/TR/REC-xml-names/`, 2006.

[33] Zoltan Somogyi, Fergus Henderson and Thomas Conway. The execution algorithm of mercury : an efficient purely declarative logic programming language. *Journal of Logic Programming*, volume 29, number 1-3 :17–64, October-December 1996.

Annexe A

Guide Utilsateur pour MSP (Anglais)

MSP Userguide

An english version of the MSP userguide

Bastien De Buyser

Mission Critical 2006

A.1 MSP template language

This document describes an XML template language based on Mercury. MSP (which stands for Mercury Server Page) requires therefore a minimum knowledge of XML (`http://www.w3.org/XML`) and Mercury (`http://www.cs.mu.oz.au/research/mercury/`).

An MSP input file is an XML document in which a target XML language is marked out by MSP tags. The target language is typically an interface one and it is merged within MSP tags in order to dynamically produce a graphical user interface according to the situation or the data retrieved.

From a more general point of vue, the motivations to work out MSP were to create an XML valid language that itself generates a result in standard XML. It was expected to manage a mix of static and dynamic elements, to be built on logical basis, to hide the complexity of the computations and to offer a generic solution by means of a simple declarative language.

MSP is based on a compiling process. From the MSP source to the XML output it passes through two compiling steps (the final output is not directly obtained from the initial file).

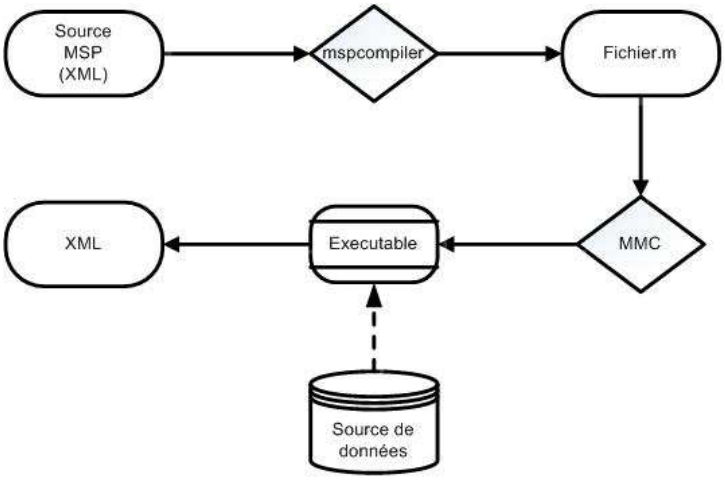


FIG. A.1 – Compiling steps

An MSP file must be compiled by mspcompiler which produces a Mercury file. Then the file.m must itself be compiled and executed in order to generate the target file in the chosen language. So MSP is aimed to finally produce XML code that could be interpreted by any browser or XML editor. It is a generic

and extensible solution for XML code generation.

A.1.1 MSP variables

This section gives an informal introduction to variables in MSP. It is difficult to describe a general rule to use them since it depends on the specific cases. The way to use variables in the different situations are expected to be understood from the examples through the document.

Assuming a variable "VAR", according to the case it must be referenced by :
- ${VAR}
- VAR

The main idea is that when wanting to refer "VAR", to write "${VAR}" (inside XML content going to be processed) will have the effect of inserting the "VAR" value in that place. However MSP instructions can require variables inside their definition, in that case (and in any case of writing in Mercury) the dollar and curly brackets must be leaved out.

The logical reasoning behind the language is operated by a Mercury program, that implies that variables must always start with a capital letter. Any expression interpretable by Mercury could actually be inserted "${"here inside"}". The use of Mercury helps to keep the declarative aspect of the template language (which was one of the MSP objectives).

A.1.2 Functionalities

MSP Functionalities correspond to specific tags in an MSP source file. They are divided into 3 categories :
- Template instructions (which are the core of the language)
- Goals (which appear in a few instructions)
- Directives

Template instructions

For each instruction a brief explaination will be given followed by one or more examples. The results of the examples are not exactly what will be produced in terms of blanks, tabulations, new lines, etc... but the semantic of the output is correct. In an MSP source, all MSP instructions must belong to the MSP namespace : `xmlns:msp="http://msp.missioncriticalit.com"`.

MSP : Forall

The MSP tag "forall" consists of two attributes : "variable" and "in_list". That list has the Mercury type "list(T)" and the variable has the type T. The variable successively takes the values of the elements from the list. The Forall instruction produces the content held between its tags for each value that the variable will take.

Example

```
<msp:forall variable="X" in_list="[1,2]">
   Some code and ${X} in the middle
</msp:forall>
```

Result

```
   Some code and 1 in the middle
   Some code and 2 in the middle
```

MSP : If

The "msp:if" is divided into three blocks : Condition, Then and Else. If the condition is true, the content between the "msp:then" will be generated and in the other case, it is the "msp:else" that will be produced. The "else" block is not compulsory in the syntax... so if it is not present it will just be interpreted as an empty "else" bloc.

Synopsis

```
<msp:if>
  <msp:condition>
     GOAL
  </msp:condition>
  <msp:then>
   What you want to generate in case of success
  </msp:then>
  <msp:else>
   What you want to generate otherwise
  </msp:else>
</msp:if>
```

NB 1 : In the condition block, the GOAL syntax must be respected (see section A.1.3 at page 21).

NB 2 : Since the Mercury stands behind MSP, its severity is to be taken into account (this remark is valid for any functionality). As an example, grounding a variable inside the condition and using it in the content will only work if : Firstly the condition succeeds (it means you are not allowed to use it inside the "else" part), secondly the variable is ground in any possible case success of the condition and thirdly, that variable has only one possible value. (The instruction is translated into a "Mercury if" thus the result can't be multideterministic).

Example

```
<msp:if>
     <msp:condition>
         <msp:and>
             <msp:mercury goal="X=1, member(X, [1,2,3])"/>
             <msp:not>
                 <msp:mercury goal="member(1, [1,2,3])"/>
             </msp:not>
             <msp:or>
                 <msp:mercury goal="member(1, [1,2,3])"/>
                 <msp:mercury goal="member(1, [2,3])"/>
             </msp:or>
         </msp:and>
     </msp:condition>
     <msp:then>
         <p>Then it succeeds and X equals ${X}</p>
     </msp:then>
     <msp:else>
         <p>Else it fails</p>
     </msp:else>
</msp:if>
```

Result

```
<p>Else it fails</p>
```

MSP : Switch

The switch enables to cover multiple cases and to generate a result according to the one encountered. It consists of multiple "msp:case" blocks each with an "msp:condition" and an "msp:content". An msp:switch will be translated into its corresponding Mercury switch. Therefore all the possible cases have to be enumerated and but they can't have overlapping scopes neither. As a consequence each solution must match exactly one case.

Example

```
<html xmlns:msp="http://msp.missioncriticalit.com" >
       <p> With the list : [yes(train), no, yes(car), yes(plane)]</p>
         <msp:forall variable="Travel"
                  in_list='[yes("train"), no, yes("car"),yes("plane")]'>
         <msp:switch>
           <msp:case>
               <msp:condition>
                  <msp:mercury goal="Travel = yes(Means)"/>
               </msp:condition>
               <msp:content>
                  <p>This one travels by ${Means}</p>
               </msp:content>
           </msp:case>
           <msp:case>
               <msp:condition>
                  <msp:mercury goal="Travel = no"/>
               </msp:condition>
               <msp:content>
                  <p>This one does not travel</p>
               </msp:content>
           </msp:case>
         </msp:switch>
         </msp:forall>
       </html>
```

Result

```
<html xmlns:msp="http://msp.missioncriticalit.com">
<p> With the list : [yes(train), no, yes(car), yes(plane)] </p>
<p>This one travels by train</p>
<p>This one does not travel</p>
<p>This one travels by car</p>
<p>This one travels by plane</p>
</html>
```

MSP : Element

This enables the creation of a tag whose name will be determined at runtime by the value of a variable. This way the name of the tag can be set dynamically. The generated tag must be a valid XML, something such as "<${VAR} >" would be refused during the parsing of the XML.

Example

```
<msp:forall variable="X" in_list='["MyElement"]'>
   <msp:element name="${X}">
      <p> I am the content of the new element </p>
 </msp:element>
</msp:forall>
```

Result

```
<MyElement>
    <p> I am the content of the new element </p>
</MyElement>
```

MSP : All

GENERAL CASE

The "All" consists of 2 blocks : A goal block (which must respect the GOAL syntax, see section A.1.3 at page 21), and a content block. Unlike the "msp:if", in the "all" case the goal can be non deterministic. The content will be repeated for each solution coming from the goal block. Multiple solutions are thus authorised but if it fails or that there are no solutions, the entire "all" instruction block will be simply ignored. Concerning variables used in the content, they must be ground (either from the goal(s) either from the upper context).

Example 1

```
<msp:all variables="X">
  <msp:goal>
     <msp:mercury goal="member(X, [1,2])"/>
  </msp:goal>
  <msp:content>
     X is ${X}
  </msp:content>
</msp:all>
```

Result 1

```
X is 1
X is 2
```

SORTING THE RESULTS

By adding an attribute to the msp:all tag, it is possible to sort the solutions in a precise order. The attributes ascending-order="..." and descending-order="..." enable to sort the solutions according to the values of a specific variable or by using a known function. It is also possible to define a sorting function via the attributes ascending-compare="..." and descending-compare="...". The above points will be demonstrated using two examples.

Example2

```
<msp:all variables="S" descending-order="length(S):int" >
   <msp:goal>
     <msp:mercury goal='member(S,
                 ["shortString",
                  "veryVery looooooooooooooooooonger string",
                  "Here is a medium string"])'/>
   </msp:goal>
   <msp:content>
     ${S}
   </msp:content>
</msp:all>
```

Result 2

```
veryVery looooooooooooooooooonger string
Here is a medium string
shortString
```

Example 3

```
<msp:all variables="X, Y"
    ascending-compare='func({X1,Y1},{X2,Y2}) =
      ( Y1 < Y2 -> (>) ; Y1 = Y2 -> (=) ; (<) )' >
  <msp:goal>
     <msp:and>
       <msp:mercury goal="member(X, [1,2])"/>
       <msp:mercury goal="member(Y, [3,4])"/>
     </msp:and>
  </msp:goal>
<msp:content>
    X is ${X} and Y is ${Y}
</msp:content>
</msp:all>
```

Result 3

```
X is 1 et Y is 4
X is 2 et Y is 4
X is 1 et Y is 3
X is 2 et Y is 3
```

WARNING : when using ascending-compare or descending-compare it's absolutely necessary to create the signature of the function according to the number of variables declared. Actually it must take in argument two Mercury tuples containing all the variables.

- `variables="X, Y"  => func({X1, Y1},{X2, Y2}) = ...`
- `variables="X, Y, Z"  => func({X1, Y1, Z1},{X2, Y2, Z2}) = ...`
- `variables="X"  => func({A},{B}) = ...`

Since the customized function is written into an xml string, XML characters must be escaped. Ex : "<" becomes "<".

MSP : Template

GENERAL CASE

There are two complementary instructions :

- msp:template
- msp:apply-template

The first enables to define a template (that is going to be translated into a Mercury function definition) and the second must be used to apply the predefined template (which corresponds to a function call).

An "msp:template" definition consists of a name, a list of parameters and a body. Each time it is called, it must be given the needed parameters, and the whole thing will be replaced by the content of its body. As for the directives (see section A.1.3 at page 24), templates must be defined right under the root tag. Once done, it is allowed to apply them anywhere inside the code. When defining a template, the name given to a parameter will be the name of its variable in the body. For each parameter defined, a (Mercury) type must be associated to it. See the examples for the precise syntax.

Note that when applying the template, the parameters can be :

- Mercury code ("${}")
- An attribute (see section A.1.2 at page 20)
- Usual content

The examples here under cover the three cases. Using "${Mercury code}" alone as a parameter value will be considered as Mercury code (so it's possible in that way to make use of pure Mercury types, see Example 1), but adding some characters (even one space) before or after the "${}" will have the effect of associating the parameter to the type "xml.content". So the definition of the types must be realised with precaution according to the use that is going to be done of them.

> NB : When applying templates, mspcompiler must know in advance which type is going to be returned. Indeed a template can return either content or attributes (see section A.1.2 at page 20) and the mix of the both types is forbidden in a same instruction (see remark Content or Attributes at section A.2.3 page 26). Since the template could be defined in an external file (see "External Files" at section A.1.2) it is not always possible to inspect its definition.

Therefore if it returns attributes it must be specified in the field "returns" like this :

```
<msp:apply-template name="some_template" returns="attributes">
```

Example 1

```
<html xmlns="http://www.w3.org/1999/xhtml"
           xmlns:msp="http://msp.missioncriticalit.com" >
  <msp:template name="my_dear_template">
     <msp:parameter name="Name" type="string"/>
     <msp:parameter name="X" type="int"/>
     <msp:parameter name="Y" type="int"/>
     <msp:body>
       <p>${Name} said that the sum of...</p>
       <p>${X} and ${Y} is <b>${X+Y}</b>.</p>
     </msp:body>
  </msp:template>
  <head>
     <title> XHTML Example </title>
  </head>
  <body bgcolor="#FFFFFF" link="#000000" text="red">
     <p>This is an XHTML example</p>
     <br/>
     <msp:forall variable="X" in_list="[1,2]">
       <p>I can count: ${X}</p>
       <msp:forall variable="Y" in_list="[3,4]">
           <msp:apply-template name="my_dear_template">
               <msp:parameter>${"You and I"}</msp:parameter>
               <msp:parameter>${X}</msp:parameter>
               <msp:parameter>${Y}</msp:parameter>
           </msp:apply-template>
       </msp:forall>
     </msp:forall>
  </body>
  </html>
```

Result 1

```
<html xmlns="http://www.w3.org/1999/xhtml"
           xmlns:msp="http://msp.missioncriticalit.com" >
  <head>
    <title> XHTML Example </title>
  </head>
  <body bgcolor="#FFFFFF" link="#000000" text="red">
    <p>This is an XHTML example</p>
    <br/>
       <p>I can count: 1</p>

       <p>You and I said that the sum of...</p>
       <p>1 and 3 is <b>4</b>.</p>

       <p>You and I said that the sum of...</p>
       <p>1 and 4 is <b>5</b>.</p>

       <p>I can count: 2</p>

       <p>You and I said that the sum of...</p>
       <p>2 and 3 is <b>5</b>.</p>

       <p>You and I said that the sum of...</p>
       <p>2 and 4 is <b>6</b>.</p>
  </body>
  </html>
```

Example 2

```
<html xmlns="http://www.w3.org/1999/xhtml"
                xmlns:msp="http://msp.missioncriticalit.com" >
   <msp:template name="my_dear_template2">
     <msp:parameter name="Content" type="list(xml.content)"/>
     <msp:parameter name="SomeAttributes" type="list(xml.attribute)"/>
     <msp:parameter name="FancyText" type="list(xml.content)"/>
     <msp:body>
        <font>
           ${Content} said that:
           <msp:forall variable="Attr" in_list="SomeAttributes">
               <msp:attribute name="${Attr ^ attr_name}"
                              value="${Attr ^ attr_value}"/>
           </msp:forall>
           ${FancyText}
        </font>
    </msp:body>
   </msp:template>
 <head>
    <title> XHTML Example </title>
 </head>
 <body bgcolor="#FFFFFF" link="#000000">
    <p>This is an XHTML example</p> <br/>
    <msp:forall variable="X" in_list='["Here is a simple text",
                    "and another one which is slighly longer"]'>
        <p>
            ${X}
            <msp:forall variable="Color" in_list='["black","red"]'>
               <msp:apply-template name="my_dear_template2">
                  <msp:parameter>
                     <b>You and I</b>
                     <em>(well, us!)</em>
                  </msp:parameter>
                  <msp:parameter>
                     <msp:attribute name="color" value="${Color}" />
                  </msp:parameter>
                  <msp:parameter>
                     the color of this text is ${Color}
              </msp:parameter>
               </msp:apply-template>
            </msp:forall>
        </p>
     </msp:forall>
  </body>
  </html>
```

Result 2

```
<html xmlns="http://www.w3.org/1999/xhtml"
         xmlns:msp="http://msp.missioncriticalit.com" >
<head>
   <title> XHTML Example </title>
</head>
<body bgcolor="#FFFFFF" link="#000000">
   <p>This is an XHTML example</p> <br/>
   <p>
      here is a simple text
      <font color="black">
         <b>You and I</b>
         <em>(well, us!)</em> said that:
         the color of this text is black
      </font>
      <font color="red">
         <b>You and I</b>
         <em>(well, us!)</em> said that:
         the color of this text is red
     </font>
  </p>
  <p>
    and another one which is slighly longer
    <font color="black">
       <b>You and I</b>
       <em>(well, us!)</em> said that:
       the color of this text is black
    </font>
    <font color="red">
       <b>You and I</b>
       <em>(well, us!)</em> said that:
       the color of this text is red
    </font>
 </p>
</body>
</html>
```

EXTERNAL FILES

Now that templates can be define and called, it would be interesting to have the possibility to create them as external files and import them when needed in one template or another. It becomes possible by using "msp:library" and "msp:import-module" (see "directives" at section 5.4 page 74).

To create such an external template is equivalent to create a Mercury library. Using "msp:library" as the root tag of your template will inform mspcompiler that it only has to treat the msp:template's under it and ignore the rest. Compiling an MSP source with the "-l" option will prevent mspcompiler from creating a function "main" in the Mercury file (See the section "options for mspcompiler"). The function "main" would be useless if the file.m is only used as a library...

Example 3 (definition)

Create the file "make_html_lists.xml" and write the following code in it :

```
<msp:library>
  <msp:template name="make_unordered_list">
    <msp:parameter name="Points" type="list(string)"/>
    <msp:body>
       <UL>
         <msp:forall variable="NewPoint" in_list="Points">
            <LI>${NewPoint}</LI>
         </msp:forall>
       </UL>
    </msp:body>
  </msp:template>
  <msp:template name="make_ordered_list">
    <msp:parameter name="Points" type="list(string)"/>
    <msp:body>
       <OL>
         <msp:forall variable="NewPoint" in_list="Points">
             <LI>${NewPoint}</LI>
         </msp:forall>
       </OL>
    </msp:body>
  </msp:template>
</msp:library>
```

Run "mspcompiler -l make_html_lists.xml", the library file make_html_lists.m is now created.

Example 3 (call)

Then create the file "test_import.xml" and write the following code in it :

```
<html \url{xmlns="http://www.w3.org/1999/xhtml"
         xmlns:msp="http://msp.missioncriticalit.com"} >
  <msp:import-module name="make_html_lists" />
  <p> The six most important animal classes are :</p>
  <msp:apply-template name="make_ordered_list">
     <msp:parameter>
         ${["Mammals", "Birds", "Fish", "Reptiles",
         "Amphibians", "Anthropods"]}
     </msp:parameter>
  </msp:apply-template>
</html>
```

Example 3 (result)

Compile it : "mspcompiler -b test_import.xml", compile then "test_import.m" and run "./test_import". You will get :

```
<html xmlns="http://www.w3.org/1999/xhtml"
         xmlns:msp="http://msp.missioncriticalit.com" >
<p> The six most important animal classes are :</p>
<OL>
   <LI>Mammals</LI>
   <LI>Birds</LI>
   <LI>Fish</LI>
   <LI>Reptiles</LI>
   <LI>Amphibians</LI>
   <LI>Anthropods</LI>
</OL>
</html>
```

NB : It could happen that multiple templates have the same name and parameters in different libraries... (that means, for Mercury speakers, more than one function having the same signature in different modules). Therefore it can be qualified by specifying the module (file.m library) which holds the targeted function with the following syntax :

```
<msp:apply-templatename="myTemplate"module="myModule">
```

For instance, modifying the previous example like this would imply specifying the module :

Example 4 (definition)

The file "make_html_lists.xml" consists only of the template "make_list" which generates an **unordered** html list.

```
<msp:library>
   <msp:template name="make_list">
      <msp:parameter name="Points" type="list(string)"/>
      <msp:body>
        <OL>
           <msp:forall variable="NewPoint" in_list="Points">
              <LI>${NewPoint}</LI>
           </msp:forall>
        </OL>
      </msp:body>
   </msp:template>
</msp:library>
```

Example 4 (call)

In "test_import.xml", define another template "make_list" which generates an **ordered** html list.

```
<html xmlns="http://www.w3.org/1999/xhtml"
         xmlns:msp="http://msp.missioncriticalit.com" >
   <msp:template name="make_list">
      <msp:parameter name="Points" type="list(string)"/>
      <msp:body>
        <UL>
          <msp:forall variable="NewPoint" in_list="Points">
             <LI>${NewPoint}</LI>
          </msp:forall>
        </UL>
      </msp:body>
   </msp:template>

   <msp:import-module name="make_html_lists" />

   <p> The six most important animal classes are :</p>
   <msp:apply-template name="make_list" module="make_html_lists">
      <msp:parameter>
           ${["Mammals", "Birds", "Fish", "Reptiles",
           "Amphibians", "Anthropods"]}
      </msp:parameter>
   </msp:apply-template>
</html>
```

Compile and run, the same result as in the example 3 will be obtained (If the unordered list is required then it is only necessery to set module="test_import" instead of "make_html_list" in msp:apply-template) :

Example 4 (result)

```
<html \url{xmlns="http://www.w3.org/1999/xhtml"
         xmlns:msp="http://msp.missioncriticalit.com"} >
<p> The six most important animal classes are :</p>
  <OL>
    <LI>Mammals</LI>

    <LI>Birds</LI>

    <LI>Fish</LI>

    <LI>Reptiles</LI>

    <LI>Amphibians</LI>

    <LI>Anthropods</LI>
  </OL>
</html>
```

MSP : Attribute

The purpose of this particular instruction is to dynamicaly add attributes to a tag. The dynamic aspect stands in the number of attributes that can be set in a tag and in the name of the attributes that can be determined by variables. Indeed, their value could already be chosen dynamically by simply using "${}" between their quotes :

```
<Tag  attribute1="${VAR1}" attribute2="${VAR2}"/>
```

Although a tag "msp:attribute" could lay under a "forall", "if","all" or "template", it will be added to the first parent tag. MSP tags are not taken into account, the attributes are only added to tags belonging to the content.

NB : "msp:attribute" could be used to instantiate parameters having the type "list(attribute)" in msp:template. (See the second example of the "template" section (A.1.2) page 13).

Example

```
<RootTag>
  <msp:forall variable="VALUE" in_list='["MyValue"]' >
    <msp:attribute name="MyAttribute" value="${VALUE}" />
  </msp:forall>
</RootTag>
```

Result

```
<RootTag MyAttribute="MyValue"/>
```

A.1.3 MSP Goal

synopsis

A GOAL consists of the following possibile constructions= :

- <msp:mercury goal="..." />
- <msp:not> GOAL </msp:not> (succeeds if GOAL fails)
- <msp:or> GOALS </msp:or> (succeeds if at least one of the GOALS succeeds, see NB*)
- <msp:and> GOALS </msp:and> (succeeds only if all the GOALS succeed)
- <msp:triple datasource="..." subject="..." predicate="..." object="..."/>

> NB* : Indeed the "or" goal (nesting multiple goals) succeeds if at least one of them succeeds but it must not be understood with the same meaning than with an imperative programming language. Since everything is going to be translated into Mercury, having more than one goal succeeding inside the "or" would make it multideterministic.

Triples retrieving

One of the objectives of MSP is to be generic, for example by enabling the RDF triple querying. This way it is possible to retrieve information from a triple data source and to exploit it inside the final XML code generated.

That functionality is part of the MSP goals but is more complex than the other ones. This section explains how to use it. Two types of RDF data sources can be queried for retrieving triples : RDF/XML files or a database via ODBC.

- <msp:triple datasource="myRdfXmlFile.xml" .../>
- <msp:triple datasource="myOdbcDataBaseName" .../>

Before requesting triples the datasources targeted must be defined with "msp:datasource" (see section 5.4 page 74). Any datasource defined can then be refered inside a query (textually, by a variable or by a mix of both).

```
Example : <msp:triple datasource="DataSourceNumber${Number}" .../>
```

The triples to retrieve will be selected by the conditions set on the subject, predicate and object.

Example

```
<Root>
   <msp:datasource type="rdf_file" source="Persons.xml" />
   This is the list of the women from PersonsRdf.xml who are married :
   <msp:all variables="X, Y">
      <msp:goal>
        <msp:and>
           <msp:triple datasource="Persons.xml"
                  subject="${X}"
                  predicate="sex"
                  object="female"/>
           <msp:triple datasource="Persons.xml"
                  subject="${X}"
                  predicate="married"
                  object="${Y}"/>
        </msp:and>
      </msp:goal>
      <msp:content>
          <p>${X} is married to ${Y}</p>
      </msp:content>
   </msp:all>
</Root>
```

Result

```
This is the list of the women from PersonsRdf.xml who are married :

              <p>carla is married to john</p>

              <p>mary is married to paul</p>
```

Triple request datasource (Persons.xml)

```
<?xml version="1.0"?>
   <rdf:RDF xmlns:rdf="http://www.w3.org/1999/XSL/Transform\#rdf">

   <person about="john">
     <personname>John Doe</personname>
     <sex>male</sex>
     <role>systemsadministration</role>
   </person>

   <person about="paul">
     <personname>Paul Smith</personname>
     <sex>male</sex>
     <role>legalsecretary</role>
     <married>mary</married>
   </person>

   <person about="mary">
     <personname>Mary Mary</personname>
     <sex>female</sex>
     <role>businessterritorymanager</role>
     <married>paul</married>
   </person>

   <person about="carla">
     <personname>Carla brownie</personname>
     <sex>female</sex>
     <role>bus driver</role>
     <married>john</married>
   </person>

   </rdf:RDF>
```

Directives

Functionalities that are neither an instruction nor a goal are directives. At the present time only two of them belongs to this category : "msp:import-module" and "msp:datasource".

ImportingModules

External modules can be imported via your MSP source. These can be existing Mercury modules or any file.m created by yourself (potentially libraries generated with the "msp:template" instruction, see section A.1.2 page 10). In order to import an external template, a tag "msp:import-module" simply must be set under the root.

RDF Data Source Declaration

The "msp:datasource" directive is aimed to declare an RDF datasource in order to enable querying on it (with <msp:triple...>). Once again datasources declarations must be direct children of the root tag, otherwise requests on it will be ignored. Inside a same template it is allowed to use multiple RDF/XML files as a source but only one ODBC source is accepted. In order to declare a file source, the corresponding path must be written in the attribute "source" with the type "rdf_file". As far as ODBC is concerned, the type must be set to "rdf_odbc" and in the field "source", the name associated to the targeted data base. In that case, to provide a userid and a password is compulsary.

- <msp:datasource type="rdf_file" source="myRdfXmlFile.xml" />
- <msp:datasource type="rdf_odbc" source="myOdbcDataBaseName" userid="MyId" pwd="ThePassword" />

A.1.4 Options for mspcompiler

Four options are available when executing **mspcompiler** :

- -h (or –help) displays the information about the options.
- -o (or –output) place the output into <file>.
- -l (or –library) tells mspcompiler that the template it has to compile is just a library and that it must not generate a "main/2". If this option isn't used and that the generated file is afterwards imported, two predicates "main/2" will exist and a compiling error will be raised. The -l flag will typically be used on an input that only consists of template definitions. However it could also be applied on a classical template... no "main" function would then be generated into the "file.m". The use of that module would then be limited to a library and could not be run itself.
- -b (or –ignore-blanks) ignore blank ctexts from the source file. That option is particulary usefull when setting some "msp:attribute" in a template parameter. The MSP compiler will ignore all the full blanks between tags if the flag "-b" is used. "Full blanks" means blanks without any content in it (no characters, no mercury code...), in that way mspcompiler sticks one against the other all the tags that have nothing between them except blanks (space, tab, carriage return). The impact is important because any blank around an "msp:attribute" tag would otherwise have been considered as content... and mixing attributes with content inside a template is forbidden.

A.2 General Remarks

A.2.1 Errors Reporting

Errors can be reported either by mspcompiler or the Mercury compiler. The MSP compiler reports mostly syntactical errors in the MSP source file. The mercury compiler can catch more complex errors, for example a missing condition in the switch. The difficulty is to link back the errors reported by the mercury compiler to the msp source file.

In order to distinguish errors origin, messages are preceded by the file name and the line number. In that way the line from the source file responsible for the error can be found back and the explaination about the error itself starts with the name of the Mercury file.

Whatever the origin, the errors messages from the second compilation are raised by the Mercury compiler. If the line number of the source file is not enough to find the cause, Mercury knowledge will be necessary to edit the file.m and try to understand the problem.

A.2.2 Querying within external templates

RDF triples querying (see MSP Goal, section A.1.3 page 21) can be realised by external templates (that means functions from an imported module). However, the data source must be defined inside the XML source.

A.2.3 Content or Attributes

Most of the instructions end up with an XML content generation. However, since "msp:attribute" enables to add an attribute to a tag, two different kind of information must be managed (an attribute doesn't represent content strictly speaking). Within a "msp:forall", "msp:if", "msp:all", "msp:switch" or "msp:template", to mix the types is prohibited. Indeed some tests[1] involving functions returning a merge of content and attributes lead to nonsense information. It was just not manageable and had no semantic. MSP instructions must therefore return either content either attributes.

[1] These tests have been carried out by other members of the MSP project team

Example of authorized use

```
<Root>
<msp:element name="NewElement" >
   <msp:forall variable='VAR' in_list='["hello","goodbye"]' >
      ${VAR}
   </msp:forall>
   <msp:forall variable='A' in_list='["small"]' >
      <msp:attribute name="size"  value="${A}" />
   </msp:forall>
</msp:element>
</Root>
```

Result

```
<Root>
  <NewElement size="small">
      hello
      goodbye
  </NewElement>
</Root>
```

A.3 Advanced Examples

The following examples require the knowledge of Mercury.

A.3.1 Handling a list in an MSP switch

MSP Source File

```
<html xmlns:msp="http://msp.missioncriticalit.com" >
<msp:template name="recursive_list_traversal">
   <msp:parameter name="L" type="list(int)"/>
 <msp:body>
     <p>The parameter list has ${length(L)} elements.</p>
     <msp:switch>
         <msp:case>
             <msp:condition>
                <msp:mercury goal="L = []"/>
             </msp:condition>
             <msp:content>
                <p>The list is empty!</p>
             </msp:content>
         </msp:case>
         <msp:case>
             <msp:condition>
                <msp:mercury goal="L = [A]"/>
             </msp:condition>
             <msp:content>
                <p>The only element is ${A}!</p>
             </msp:content>
         </msp:case>
         <msp:case>
             <msp:condition>
                <msp:mercury goal="L = [A,B|R]"/>
             </msp:condition>
```

```
                <msp:content>
                    <p>The two first elements are
                       ${A} and ${B}.</p>
                    <msp:apply-template name="recursive_list_traversal">
                        <msp:parameter>${R}</msp:parameter>
                    </msp:apply-template>
                </msp:content>
            </msp:case>
        </msp:switch>
     </msp:body>
  </msp:template>
  <head> </head>
  <body>
   <msp:apply-template name="recursive_list_traversal">
       <msp:parameter>
          ${[1,2,4,5,7,11,13,17,19,23,27,29, 31]}
       </msp:parameter>
   </msp:apply-template>
  </body>
  </html>
```

Result

```
<html xmlns:msp="http://msp.missioncriticalit.com" >
<head/>
<body>
    <p>The parameter list has 13 elements.</p>
    <p>The two first elements are 1 and 2.</p>
    <p>The parameter list has 11 elements.</p>
    <p>The two first elements are 4 and 5.</p>
    <p>The parameter list has 9 elements.</p>
    <p>The two first elements are 7 and 11.</p>
    <p>The parameter list has 7 elements.</p>
    <p>The two first elements are 13 and 17.</p>
    <p>The parameter list has 5 elements.</p>
    <p>The two first elements are 19 and 23.</p>
    <p>The parameter list has 3 elements.</p>
    <p>The two first elements are 27 and 29.</p>
    <p>The parameter list has 1 elements.</p>
    <p>The only element is 31!</p>
</body>
</html>
```

A.3.2 Handling a list with a template

To use "msp:template" more than one time with the same name will create a Mercury function with multiple clauses. So the different cases of a list can be handled by defining each of them in a different "msp:template".

MSP Source File

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <msp:template name="my_dear_template">
    <msp:parameter name="[]" type="list(int)"/>
    <msp:body>
       <p>The list is empty. That's a shame!</p>
    </msp:body>
  </msp:template>
  <msp:template name="my_dear_template">
    <msp:parameter name="[X|Lx]" type="list(int)"/>
    <msp:body>
       <p>The list has ${length([X|Lx])} elements.</p>
    </msp:body>
  </msp:template>
  <head>
     <title> XHTML Example </title>
  </head>
  <body bgcolor="#FFFFFF" link="#000000" text="red">
    <p>This is an XHTML example</p>  <br/>
    <msp:forall variable="X" in_list="[[1], [1,5,7],[], [0,13,17]]">
       <msp:apply-template name="my_dear_template">
           <msp:parameter>${X}</msp:parameter>
       </msp:apply-template>
    </msp:forall>
  </body>
</html>
```

Result

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title> XHTML Example </title>
</head>
<body bgcolor="#FFFFFF" link="#000000" text="red">
  <p>This is an XHTML example</p> <br/>
     <p>The list has 1 elements.</p>
     <p>The list has 3 elements.</p>
     <p>The list is empty. That's a shame!</p>
     <p>The list has 3 elements.</p>
</body>
</html>
```

A.3.3 Summary Example

This is an example mixing a bit all the functionalities.

MSP Source File

```
<Royalties xmlns:msp="http://msp.missioncriticalit.com" >
  <msp:datasource type="rdf_file"
                  source="../tests/rdf/SetOfPeople1.xml" />
  <msp:datasource type="rdf_file"
                  source="../tests/rdf/SetOfPeople2.xml" />
  <msp:datasource type="rdf_file"
                  source="../tests/rdf/SetOfPeople3.xml" />
  <msp:import-module name="royalties2xml" />
  <msp:apply-template name="summary_template" />
</Royalties>
```

royalties2xml : XML source of the imported file

```
<Root xmlns:msp="http://msp.missioncriticalit.com">
  <msp:template name="summary_template">
    <msp:body>
       <msp:forall variable="Num" in_list="[1,2,3]" >
         <msp:all variables="Name, Place, Status"
                  descending-order="length(Status):int">
            <msp:goal>
               <msp:and>
                 <msp:triple datasource=
                    "../tests/rdf/SetOfPeople${int_to_string(Num)}.xml"
                    subject="${Name}"
                    predicate="sex"
                    object="male"/>
                 <msp:triple datasource=
                    "../tests/rdf/SetOfPeople${int_to_string(Num)}.xml"
                    subject="${Name}"
                    predicate="rank"
                    object="royalty"/>
                 <msp:triple datasource=
                    "../tests/rdf/SetOfPeople${int_to_string(Num)}.xml"
                    subject="${Name}"
                    predicate="place"
                    object="${Place}"/>
                 <msp:triple datasource=
                    "../tests/rdf/SetOfPeople${int_to_string(Num)}.xml"
                    subject="${Name}"
                    predicate="role"
                    object="${Status}"/>
                    <msp:or>
                        <msp:mercury goal='Name = "Philippe"'/>
                        <msp:mercury goal='Name = "Laurent"'/>
                        <msp:mercury goal='Name = "Charles"'/>
                        <msp:mercury goal='Name = "Albert II"'/>
                    </msp:or>
               </msp:and>
            </msp:goal>
            <msp:content>
               <msp:element name="${Status}" >
                 <msp:attribute name="from" value="${Place}" />
                 <msp:if>
                    <msp:condition>
                      <msp:mercury goal='member(Num, [1, 3])'/>
                    </msp:condition>
                    <msp:then>
```

```
            ${Status} ${Name} belongs to the set of people number ${Num}
                      </msp:then>
                      <msp:else>
            ${Status} ${Name} belongs for sure to the set of people number 2
                      </msp:else>
                   </msp:if>
                </msp:element>
             </msp:content>
            </msp:all>
          </msp:forall>
      </msp:body>
    </msp:template>
  </Root>
```

Result

```
<Royalties xmlns:msp="http://msp.missioncriticalit.com" >
  <Prince from="Belgium">
    Prince Philippe belongs to the set of people number 1
  </Prince>
  <Prince from="Belgium">
    Prince Laurent belongs for sure to the set of people number 2
  </Prince>
  <Prince from="Wales">
    Prince Charles belongs to the set of people number 3
  </Prince>
  <King from="Belgium">
    King Albert II belongs to the set of people number 3
  </King>
</Royalties>
```